ADVANCED SOFTWARE COMPREHENSION TECHNIQUES

Habilitation Thesis

Andreas Bollin
University of Klagenfurt
Institute of Informatics Systems
Software Engineering and Soft Computing

A UNIVERSITY OF KLAGENFURT PUBLICATION

Published by University of Klagenfurt, Klagenfurt, Austria.

To my beloved wife and my
wonderful children

CONTENTS

PREFACE

Today's technical systems become more and more software-intensive which typically means that software is the major component that provides the needed functionality. For example, nowadays 50% of the total costs of the development of an airplane go into software development. Moreover, only 6% of these expenses are spent on the actual development of the software system, and about 44% are consumed by testing, verification and validation activities. Consequently, considerations concerning reliability and dependability become more and more important, and formal methods offer themselves back to the software engineering community.

The use of formal specifications seem to be a silver bullet, but there are a couple of drawbacks. At first, not all stakeholders are familiar with the formal notations, and agreeing upon assertions in specifications (and thus agreeing upon parts of the requirements) is complicated. Secondly, the mathematically dense notations, combined with the inherent complexity of the system, might be impediments when trying to understand the specified system – even for experts. Finally, it takes a lot of effort to keep specifications up-to-date during the development live-cycle. When becoming acquainted with the requirements, and with continually increasing project knowledge, new versions of the specifications are created. But with this, specifications evolve over time. Without any guidance, developers do not know whether the overall quality increases or eventually decreases.

The habilitation thesis focuses on these impediments and consists of four parts.

The first part, "Specification Comprehension", discusses typical impediments when trying to understand formal specifications and offers a gentle introduction to the field of software comprehension. It motivates for an improved comprehension support, introduces the research context and summarizes my own contributions to this area of research.

The second part, “Assessment”, deals with the aspect of specification metrics and measurements. It addresses the fact of software evolution in respect to formal specifications, presents a measurement system for focusing on complexity and quality attributes and demonstrates how to make use of these measures.

The third part, “Specification Visualization”, deals with the issue of the transformation of formal specifications. In order to support the different stakeholders in a project, formal specifications are enriched by other (semi-formal) graphical notations. As UML is widespread, it has been chosen as the target notation for the transformation process, and the first part of this thesis presents approaches to produce optimized static, as well as dynamic UML models from formal specifications.

The fourth part, “Concept Location and Management”, deals with the maintenance support for large and complex formal specifications. It presents a concept location model, the necessary tool-sets for the related comprehension tasks and also a system for storing and retrieving concepts into and from a database.

A. BOLLIN

Klagenfurt, May 2012

ACKNOWLEDGMENTS

From Prof. Peter Lucas, Professor Emeritus at the Institute of Software Technology, Graz University of Technology, Austria, I once received the initial spark to work with formal specifications. He told me that, as a scientist, we will have reached the Olympus of software (system) development only when we are able to deliver all our products with a guarantee certificate instead of supplementing them with license agreements providing (nearly) no guarantee. In his (and also mine) opinion this goal can only be reached by making formal methods popular again.

Prof. Roland Mittermeir, Professor at the Institute of Informatics Systems, University of Klagenfurt, Austria, who was also my doctoral advisor, encouraged me to keep on working on this topic, too. My conviction is that formal software development becomes a widely appreciated method which is broadly used by students as well as by developers in order to make high quality systems happen. To make this vision come true, working with formal specifications has to be as easy as working with traditional programming languages in their state-of-the practice programming environments. Prof. Mittermeir taught me the arsenal (and patience) needed for working on this scientific topic, situated on the edge between the formal methods and the software development community.

Last but not least, I have to mention those important people without whose support, also during my sickness, it would not have been possible to finish this work: my wife Katalin and the parents of my God-child, Sonja and Heinz.

To all these wonderful people I owe a deep sense of gratitude especially now that the project of working on the habilitation thesis has been completed.

A. B.

PART I

SPECIFICATION COMPREHENSION

CHAPTER 1

INTRODUCTION

1.1 Research Context and Motivations

Formal methods and the use of formal specification languages play a crucial role in software engineering. They enable verification and validation of SW designs and form the basis for refinement steps [58]. Already in 1996, Clarke and Wing [19] presented a comprehensive survey of the use of formal specifications and provided over 120 references to successful projects and tools. In 2002, Sobel and Clarkson [52] then demonstrated that a formal analysis is of great benefit to the implementation with respect to completeness. They showed that development groups using formal methods pass nearly 100% of a standard set of test cases in comparison to 45.5% passed by control teams not using formal methods.

The argument of quality is not just an academic one. Formal methods are used in practice as companies using formal methods successfully demonstrate [51]. Additionally, formal modeling is not as inflexible as one might believe. Changing requirements involve more agile processes and it is exciting to see that formal methods and agile software development processes can be combined effectively [3]. The list of formal specification projects can easily be continued and a comprehensive collection is found at the Formal-Methods-Wiki page maintained by Jonathan Bowen [14].

It looks like a silver bullet, but the use of formal specifications comes with a price. Apart from several myths that are around [26, 15], specifications are legitimately criticized to be hard to understand due to their semantical compactness. The (dense) mathematical notation and missing redundancy allows for writing down projects' concepts and requirements by

just a few lines of specification text. However, as projects are inherently complex, a single specification easily sums up to thousands of lines of specification text. For example, the Tokeneer specification [21] – a specification for an identification station consisting of a fingerprint reader, a display and a card reader – contains over 4,800 lines of Z code and more than 140,000 dependencies among its predicates. It is clear that such specifications are hard to understand. Without proper (tool) support such specifications are not easy to handle.

One advantage of a formal specification is that it specifies the requirements in a formal manner. The overall objective is to ensure that everything has been written down in an unambiguous manner. Furthermore, a specification can be checked formally by tools, proofing specific properties of it. As a specification is meant to be a precise description of the future system, it is an important artefact for the validation of the requirements. But, and this is the point, this is true only when everyone understands the specification. When not all of the relevant stakeholders of a project are familiar with the notation then the reflection on the semantics of the specification becomes much more difficult.

The impediments mentioned above (complexity, missing redundancy and unfamiliar notation) lead to the belief that specification languages are "write-only languages" and specifications are just written once at the initial phases of a project [40]. In fact, formal specifications only remain valid artifacts in the life-cycle when they are kept up to date during *all* evolutionary steps. As a consequence, specifications have to be adapted whenever requirements are better understood, and they have to be kept consistent with the code during all different maintenance activities. The formal development of the Web Service Definition Language 2.0, its 139 revisions are publicly available in a CVS system [18], shows how a specification evolves and that a lot of effort is necessary to keep it useful. Specifications deteriorate over time, and without any measures and tools changes in the specification text are hard to assess. Unfortunately, the awareness of the importance of measuring and predicting quality related properties is growing only slowly.

To summarize, the following characteristics of formal specifications impede the use of formal methods in software development, increase their perceived complexity, and lead to a variety of myths and obstacles that I want to address with my research:

- The use of "fancy" mathematical symbols. Not all stakeholders of a project are familiar with the notations used. Whilst quite big portions of the notations of model-oriented approaches (like Z [53] or VDM [32]) can be understood with sufficient basic understanding of mathematics, the situation is quite different with temporal logic notations where modal operators are used. But, irrespective of whether the notation is known or not, the mathematical notations are far away from the (semi-formal) notations developers are currently using.
- Too few clues for reconstructing the behavior. At least with small programs the well defined execution sequence among statements allows for partial comprehension. A desk-check in the form of a program "dry run" with some assumed values is easily possible. Trying to comprehend the program without assuming specific values bound to the program's variables is much harder. Due to the declarative nature of specifications, the writer does not need to worry about the order of execution, but this also means that one no longer has that built-in clue of "dry-runs" for partial comprehension. Specification animation might overcome this situation, but an animation of state-changes can not be compared to the knowledge gained by program execution during testing.

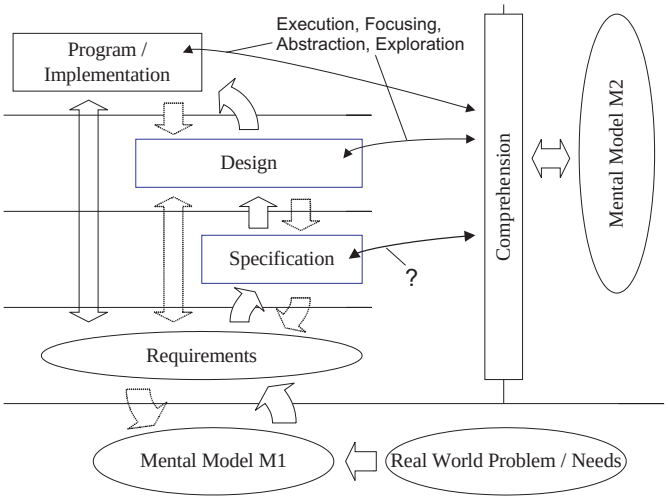


Figure 1.1 Several artefacts can be referred to in order to understand a complex system. Besides design documents, formal specifications are also sources for development and maintenance phases. But, for their understanding special skills are necessary [7, p.75].

- Too few clues for reconstructing the structure of the system to be created. Putting too much structure into a specification is usually understood to be a hint towards implementation and thus avoided. Some notations provide abstraction techniques on the granularity level of objects, but they are of no use when specifications are getting really big. Specifications focus on properties and not so much on structure.

- Size-complexity due to the compact representation. There are many notions of complexity, and some of them can not be avoided (for example the environmental or application domain complexity). However, according to Alagar and Periyasami [1], size complexity is probably the most critical type of complexity. Leveson [38] also argues that almost all accidents with software involved are due to this kind of complexity. Specifications, when printed, might not be as long as their program counterparts, but due to their semantical compactness they are not less complex.

- Complexity due to (hidden) dependencies. This characteristic goes hand in hand with the complexity of size. The number of predicates in a specification can make understanding a big issue, but the number of dependencies between them is often so high that only tool support can help. The Elevator specification [16], a simple textbook example of 6 pages in print and 190 lines of specification text, already contains more than 2,000 dependencies among the predicates (see 12.3, Figure 12.3 for an overview).

- Sensitiveness to evolutionary effects. Specifications age when they are modified. This change can be an improvement but might also lead to a decline in quality. Here, the popular quote “when you do not know where you are, any way will do” also holds. Only with a suitable measurement system that is constantly used (and refined), one will be able to react to evolutionary changes correctly.

There are many benefits of the use of formal specifications, but the above characteristics are part of the reasons why developers (and managers) shy away from using them.

The objective of my research is to demonstrate that it is possible to deal with all of the impediments and to bring back the use of formal methods to curriculas and, finally, to the industry. Figure 1.1 presents a simple model of a (Waterfall-like) development process. It stresses the fact that a mental model $M2$ is to be reconstructed by several steps commonly known as software comprehension activities. Ideally, the models $M1$ and $M2$ are largely identical. Formal specifications are a great basis to achieve a high congruence of $M1$ and $M2$. They are closer to the requirements and would be ideal candidates if there would be a better comprehension support. Getting my motivation to the heart of the matter, I want to define (and finally extinguish) the question mark in Figure 1.1.

In working towards this goal, the contributions make use of, map and introduce comprehension techniques to formal specifications. The following section motivates the three main fields of my research and summarizes my most important contributions to this field.

1.2 Contributions

According to Tilley [55], three activities are typical for (program) comprehension processes:

A1 Data gathering using static analysis of code or using dynamic analysis based on the execution of the program.

A2 Information exploration and visualization using navigation aids, analysis tools and different types of presentation.

A3 Knowledge organization by focusing and creating abstractions for efficient storage and retrieval.

As discussed in my PhD thesis [4, p.26], these activities can also be mapped to the field of specification comprehension. However, the focus of the thesis was on specification slicing. What remains is further support for activities [A1] to [A3]: information has to be presented in an understandable manner, and knowledge has to be created and organized. Furthermore, data about the specifications has to be gathered. The following collection of papers puts the focus on my contributions with respect to these three main research directions, the **Assessment of Formal Specifications**, Specification **Visualization** and Specification **Concept Location and Management**.

1.2.1 Assessment of Formal Specifications

In the field of software development quality indicators and complexity metrics are commonly used. On the other hand, rather rarely one speaks (or even thinks) about the quality or complexity of specifications – quality is inherently assumed and complexity is accepted. But, only with a reasonable definition and use of measures the following important questions can be answered:

M1 Is it possible to make predictions about the quality and complexity of a specification?

M2 Is it possible to keep (or even raise) the quality of a specification?

M3 Can we detect deterioration?

M4 Is it possible to make predictions about the expenditure (the costs) of a possible implementation?

Papers that deal with the measurement of specifications are indeed rare. An overview of approaches that make use of specification metrics can be found in Chapter 4.7. However, with the exception of a few approaches all of them just make use of size-based measures (like counting lines of specification text or the number of predicates). In 2004, I suggested to make also use of dependency information and thus to extent the set of existing complexity measures [4, pp.163ff]. The usability of these measures was shown by example (see Chapter 12 for details), but still a lot of open questions remained:

M5 What could be a quality indicator for formal specifications?

M6 How good (or unique) are the measures?

M7 Do the measures reflect properties similar to their program-related counterparts?

For sure the list of questions can be continued. However, by answering the questions M1 to M7, one can help to establish creating formal specifications as a core deliverable in the software development process. For reaching this goal, I can list the following contributions:

1. Introduction and definition of quality-based formal specification measures [9] (Chapter 2). This paper provides answers to questions M5 and M7. It maps the idea of slice-based coupling and cohesion metrics to formal specifications and demonstrates their sensitiveness in respect to different maintenance activities. It also shows that these measures have a similar behavior than comparable measures for program-related counterparts.

2. Introduction of a refined model of SW-evolution that includes formal specifications [11] (Chapter 3). This paper demonstrates that specifications are not an exception for aging. It makes use of the measures introduced in Chapter 2 and applies them to all 139 versions of the Web Service Definition Language (WSDL) specification. It demonstrates that the measures are very useful in detecting changes and that specifications indeed can deteriorate. With an extension of the "versioned staged model" [2] it thus provides answers to questions M1, M2 and M3.

3. Evaluation of complexity- and quality-based measures concerning their expressiveness [12] (Chapter 4). This journal paper is partially an extended version of the papers in Chapter 2 and 3, but it additionally presents a large study demonstrating that the measures describe unique properties of specifications. Based on a collection of sample specifications (totally more than 20,000 lines of specification text) it compares size-, structure- and semantics-based measures head to head and gives a summary of their means and standard deviations. Apart from again dealing with questions M1 to M3, it provides detailed answers to questions M5 and M6.

4. Demonstrating the relationship between specification- and implementation-based measures [13] (Chapter 5). What is left is the answer to the question whether implementation-related measures can be predicted or not. For this, the paper takes a closer look at the Tokeneer specification and its implementation in Ada [21]. It analyzes the relationships between specification and implementation-based measures, and, finally,

comes up with a prediction model. As such, it provides answers to questions M4 and M7.

In Part V there are two additional papers that focus on continuing aspects of my PhD thesis. The first paper [6] (Chapter 11) has been included as it provides a good introduction to the field of specification maintenance and how the identification of specification abstraction (fragments, slices, chunks) is applicable to Z specifications. The second paper [5] (Chapter 12) provides a good introduction to the definitions and applications of complexity measures for formal Z specifications.

1.2.2 Specification Visualization

Similar to the classifications provided for program comprehension, specification visualization tools and approaches can be assigned to the following three classes [41]:

SV1 This class of specification comprehension tools aims at rewriting a specification and/or reasoning about it. Usually the level of abstraction is changed or raised.

SV2 This class makes use of animation (interpretation and execution) as the next step to render formal specifications "digestible".

SV3 This class comprises tools for writing, reading and browsing specifications. It deals with the proper visualization of the needed information.

The support for all three classes is, in my opinion, still not satisfactory (from the perspective of different stakeholders). Environments like Ergo [57, 56], Isabell [34, 44] or Overture [45] are very useful, but rewriting and formally reasoning about specifications need special skills and are a cost- and resource-intensive task. Animations, as for example implemented in Alloy [31], allow for posing "questions" that can be answered "automatically". This approach can also be seen as a low-cost alternative to formal proofs and is, at least, due to a graphical feedback, easier to communicate to stakeholders. The tools for writing/editing specifications are mature, but visualization is, when not directly used for specifying classes, still limited to pretty-printing or to an outline of the general structure of the specification text.

To summarize, specification visualization techniques do not make use of widely-understood notations like UML. Accounting for the situation, I can list the following contributions:

1. Mapping a formal specification to UML diagrams [7] (Chapter 6). By the example of Z I provide two mapping strategies that produce static as well as dynamic views onto specifications. A mapping to class diagrams has already been defined for specification languages others than Z [24, 29, 30], but up to then, only static aspects have been regarded. With the reconstruction of control- and data dependencies in formal specifications (see Chapter 11 for more information about the approach), a dynamic mapping is possible, too. Control dependencies between two operations are mapped to activity diagrams with control-flow vertices, and data dependencies are mapped to activity diagrams by using object flow nodes with the label of the relevant identifier(s). Details can be found in Chapter 6. Both types of diagrams combined provide a good picture of what "statics" and "dynamics" is in a formal specification.

2. Optimized mapping of formal specifications to UML diagrams [10] (Chapter 7). Apart from a potential loss in semantic expressiveness, existing mapping strategies have to deal with the problem of assigning operations to the most suitable classes. Idani and Ledru [29] suggest to identify relevant classes (called pertinent classes) by taking the numbers of calls and references to identifiers into account. It is possible to go even one step further. By regarding inter-schema flow measures I suggest to balance the values for inter-schema coupling for all methods within and between UML classes. By using the notion of coupling, two benefits arise: the mapping is based on an intuitively comprehensible measure and the resulting UML diagrams are optimized in respect to quality considerations. Details about the definitions of the measures and a valid example can be found in Chapter 7.

1.2.3 Concept Location and Management

The term "concept location" denotes a process that maps domain concepts (patterns, features) to code positions [37]. The technique itself usually follows an intuitive process. Only when experience and familiarity with the code under investigation is not sufficient, more structured approaches like pattern matching, dynamic analysis and static analysis are needed.

The location of concepts plays an increasing role in software maintenance as requests are often formulated in terms of concepts and do not necessarily contain references to the lines of code (or text) to be altered. With the necessity of keeping specifications and code consistent, change requests also affect parts of the specifications. The problem stays the same: how to support the comprehension process of (complex) formal specifications.

Code maintenance and reverse engineering activities are supported by tools and frameworks. There are:

RE1 SW-engineering frameworks that can be used for reverse engineering activities [50, 23, 43, 33, 39]. They enable the reconstruction (or extractions) of diagrams from code, establish links between the different representations and support round-trip engineering. However, they assume that the code has already been written within the framework or they are limited to specific notations.

RE2 Explicit reverse engineering tools. The input is the code or an abstraction of it. The tools then sustain the process of creating extractions or views onto the source. Popular representatives are $RIGI$ [42, 54], going back to the work of Müller, Storey, Tilley, and Wong in the early 90s, or $Bauhaus$ [36]. In the meantime there exists a lot of extensions for this class of reverse engineering tools, especially for C++ and Java programs [25, 35, 20, 28, 22, 27].

RE3 Frameworks that are designed for concept location approaches [17, 59, 49]. They make use of techniques similar to those of the above mentioned reverse engineering environments, but provide additional support for storing and retrieving the identified concepts.

Unfortunately, none of the above mentioned tools or frameworks provides support for formal specifications. Environments for formal specifications have their focus on writing syntactically correct specifications and, eventually, they provide tools for verifying them.

Accounting for this situation, I worked towards a specification support covering the tasks of all three types of frameworks. For RE1 there is visualization support available as described in Chapters 6 and 7. For RE2 and RE3 there is now also support for generating specification abstractions, including storage and retrieval. I can list the following contributions:

1. Introducing an approach that facilitates the comprehension process of complex formal specifications [8] (Chapter 8). This contribution consists of two parts. At first, a process model for specification comprehension is presented. Secondly, an iterative approach for specification concept identification is introduced, aiming at exactly those cases when not even a surmise can be made about the concepts' location. For the second part, clustering techniques are introduced for formal Z specifications. The necessary definitions, the clustering algorithm and an example of its use can be found in Chapter 8.

2. Presenting an approach for persistently storing formal specification concepts in a database [48, 47] (Chapters 9 and 10). The identification of concepts is seen as a multidimensional problem, consisting of a syntactic representation level, a semantic model level, and a semantic concept level[1]. This model has been mapped to a database schema and forms the basis for the concept location process. The environment has been implemented during the Master thesis of Pohl [46]. The ideas behind the approach are summarized in the paper of Chapter 9, and performance and time-complexity considerations are to be found in the paper of Chapter 10.

1.3 Full Reference of the Collection Papers

Part II: Assessment

`Chapter 2:`
Andreas Bollin. Slice-based Formal Specifiation Measures – Mapping Coupling and Cohesion Measures to Formal Z. In Cèsar Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium*, NASA/CP-2010-216215, pages 24–34. NASA, Langley Research Center, April 2010.

`Chapter 3:`
Andreas Bollin. Is There Evolution Before Birth? Deterioration Effects of Formal Z Specifications. *Lecture Notes in Computer Science*, 6991 (Formal Methods and Software Engineering): 66–81, 2011.

`Chapter 4:`
Andreas Bollin. Metrics for Quantifying Evolutionary Changes in Z Specifications. *To appear in Software Maintenance and Evolution: Research and Practice*. Wiley and Sons Ltd., 28 pages, Accepted June 2012.

[1]For example, the calculation of a specification or program slice (stored in the concept level) depends on the concept of dependencies (model level), the concept of scope (model level), and the basic elements in the source (representation level). All these concepts (at different levels) are calculated and stored in a MySQL database for later use.

Chapter 5:
Andreas Bollin and Abdollah Tabareh. Predictive Software Measures based on Z Specifications - A Case Study. *Submitted to the 2nd Workshop on Formal Methods in the Development of Software, co-located with the 18th International Symposium on Formal Methods, CNAM Paris, France*, 8 pages, August 2012.

Part III: Specification Visualization

Chapter 6:
Andreas Bollin. Crossing the Borderline – from Formal to Semi-Formal Specifications. *IFIP Journal on Software Engineering Techniques: Design for Quality. Springer, 227:*73–84, 2006.

Chapter 7:
Andreas Bollin. *Coupling-based Transformations of Z Specifications into UML Diagrams*. *International NASA Journal on Innovations in Systems and Software Engineering*, 7(4):283-292, 2011.

Part IV: Concept Location and Management

Chapter 8:
Andreas Bollin. Concept Location in Formal Specifications. *Journal of Software Maintenance and Evolution – Research and Practice*, 20(2):77–105, 2008.

Chapter 9:
Daniela Pohl and Andreas Bollin. Concept Management: Identification and Storage of Concepts in the Focus of Formal Z Specifications. *Communications in Computer and Information Science. Springer*, 69(II):248–261, 2010.

Chapter 10:
Daniela Pohl and Andreas Bollin. Database-Driven Concept Management – Lessons Learned from Using EJB Technologies. In *4th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 227–238. IEEE Computer Society, May 2009.

Part V: Further Reading

Chapter 11:
Andreas Bollin. Maintaining Formal Specifications. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary*, pages 442–453. IEEE Computer Society, 2005.

Chapter 12:
Andreas Bollin. The Efficiency of Specification Fragments. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering, Delft, The Netherlands*, pages 266–275. IEEE Computer Society, 2004.

1.4 Remarks on the Papers

Each of the following chapters reproduces the content of a paper without any change in their substance from its published version. There are 5 journal papers (1 is under review),

and 4 conference papers (1 submitted and 3 already appeared). Two additional conference papers (that of the $WCRE$ 2004 and $ICSM$ 2005) are added to Part V of this thesis, but they are based on outcomes of my PhD thesis. The following issues about the papers are worth being mentioned:

- All the papers cover some of my research contributions and contain approaches developed by myself. They also express my personal opinion regarding the different topics.
- None of the papers or results (except those in Part V) are part of my PhD thesis. For some of the experiments a software environment has been used that was originally designed for my PhD thesis. However, it had to undergo major extensions to be used as described in the papers.
- All the papers where I am the sole author, were entirely written by myself. In those papers where I am co-author (in Chapters 3, 5 and 9) most parts of the content have been written by myself, based on joint work and discussion.
- Due to the different styles that are used at conferences and journals, the hereinafter published papers do not appear in their original formats. They have been reformatted when necessary without alterations to their contexts. Original copies of the papers can be requested from the author.

REFERENCES

[1] V.S. Alagar and K. Periyasamy. *Specification of Software Systems*. Springer, 1998.

[2] Keith Bennet and Vaclav Rajlich. Software Maintenance and Evolution: a Roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–89. ACM New York, NY, USA, 2000.

[3] Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal Versus Agile: Survival of the Fittest. *IEEE Computer*, 42(9):37–54, September 2009.

[4] Andreas Bollin. *Specification Comprehension – Reducing the Complexity of Specifications*. PhD thesis, AAU Klagenfurt, April 2004.

[5] Andreas Bollin. The Efficiency of Specification Fragments. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering, Delft, The Netherlands*, pages 266–275. IEEE Computer Society, 2004.

[6] Andreas Bollin. Maintaining Formal Specifications. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary*, pages 442–453. IEEE Computer Society, 2005.

[7] Andreas Bollin. Crossing the Borderline – from Formal to Semi-Formal Specifications. *IFIP Journal on Software Engineering Techniques: Design for Quality. Springer*, 227:73–84, 2006.

[8] Andreas Bollin. Concept Location in Formal Specifications. *Journal of Software Maintenance and Evolution – Research and Practice*, 20(2):77–105, 2008.

[9] Andreas Bollin. Slice-based Formal Specifiation Measures – Mapping Coupling and Cohesion Measures to Formal Z. In Cèsar Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium*, NASA/CP-2010-216215, pages 24–34. NASA, Langley Research Center, April 2010.

[10] Andreas Bollin. Coupling-based Transformations of Z Specifications into UML Diagrams. *International NASA Journal on Innovations in Systems and Software Engineering*, 7(4):283–292, 2011.

[11] Andreas Bollin. Is There Evolution Before Birth? Deterioration Effects of Formal Z Specifications. *Lecture Notes in Computer Science*, 6991(Formal Methods and Software Engineering):66–81, 2011.

[12] Andreas Bollin. Metrics for Quantifying Evolutionary Changes in Z Specifications. *To appear in Software Maintenance and Evolution: Research and Practice. Wiley and Sons Ltd.*, 28 pages, Accepted June 2012.

[13] Andreas Bollin and Abdollah Tabareh. Predictive Software Measures based on Z Specifications - A Case Study. In *Submitted to the 2nd Workshop on Formal Methods in the Development of Software, co-located with the 18th International Symposium on Formal Methods, CNAM Paris, France*, 8 pages, August 2012.

[14] Jonathan P. Bowen. Formal Methods Wiki. http://formalmethods.wikia.com. Page last visited Oct 9th, 2010, February 2010.

[15] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, July 1995.

[16] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[17] Kunrong Chen and Vaclav Rajlich. RIPPLES: Tool for Change in Legacy Software. In *IEEE International Conference on Software Maintenance*, page 230, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[18] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0. http://www.w3.org/TR/wsdl20, 2007.

[19] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions, CMU Computer Science Technical Report CMU-CS-96-178. Technical report, Carnegie Mellon University, August 1996.

[20] Computer Human Interaction and Software Engineering Lab (CHISEL). SHriMP Homepage. www.thechiselgroup.org/shrimp, Page last visited: April 2012.

[21] David Cooper and Janet Barnes. Tokeneer ID Station – EAL5 Demonstrator: Summary Report. S.p1229.81.1, Praxis High Integrity Systems, 2008.

[22] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO – Generic Understanding of Programs – An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[23] Eclipse-GMT. Homepage. www.eclipse.org/gmt/, Page last visited: April 2012.

[24] Houda Fekih, Leila Jemni, and Stephan Merz. Transformation des spècifications B en des diagrammes UML. In *Proceedings of Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL'*04, pages 131–148, 2004.

[25] R. Ferenc, A. Beszedes, M. Tarkiainen, and T. Gyimothy. Columbus – Reverse Engineering Tool and Schema for C++. In *IEEE International Conference on Software Maintenance*, pages 172–181, Montreal, Canada, 2002.

[26] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[27] Ric Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. GXL Graph Exchange Library Homepage. http://www.gupro.de/GXL/, Page last visited: April 2012.

[28] Rick Holt. PBS – The Portable Bookshelf Homepage. www.swag.uwaterloo.ca/pbs, Page last visited: April 2012.

[29] Akram Idani and Yves Ledru. Object Oriented Concepts Identification from Formal B Specifications. In *Formal Methods in Industrial Critical Applications, FMICS'04*, 2004.

[30] Akram Idani, Yves Ledru, and Didier Bert. Derivation of UML Class Diagrams as Static Views of Formal B Developments. In *7th International Conference on Formal Engineering Methods, ICFEM 2005*, pages 37–51, 2005.

[31] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. The MIT Press, Cambridge, Massachusetts, 1996.

[32] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, second edition, 1990.

[33] F. Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA 2005), Workshop on Traceability*, 2005.

[34] Kolyang, T. Santen, and B. Wolff. A Structure Preserving Encoding of Z in Isabelle/HOL. In *Theorem Proving in Higher Order Logics - 9th International Conference*. Springer Verlag, 1996.

[35] E. Korshunova, M. Petkovic, M. G. J. van den Brand, and M. R. Mousavi. CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code (Tool Paper). In *Working Conference on Reverse Engineering (WCRE'06)*, Benevento, Italy, 2006.

[36] Rainer Koschke. Software Visualization for Reverse Engineering. *Lecture Notes in Computer Science*, 2269:524–527, 2002.

[37] Wojtek Kozaczynski, Jim Ning, and Andre Engberts. Program Concept Recognition and Transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, December 1992.

[38] Nancy G. Leveson. Software Safety in Embedded Systems. *Communications of the ACM*, 34(2):35–46, 1991.

[39] MetaEdit+. Homepage. www.metacase.com, Page last visited: March 2009.

[40] Roland Mittermeir and Andreas Bollin. Demand-Driven Specification Partitioning. In László Böszörmenyi and Peter Schojer, editors, *Modular Programming Languages*, volume 2789 of *Lecture Notes in Computer Science*, pages 241–253. Springer Berlin / Heidelberg, 2003.

[41] Roland T. Mittermeir, Andreas Bollin, Heinz Pozewaunig, and Dominik Rauner-Reithmayer. Goal-Driven Combination of Software Comprehension Approaches for Component Based Development. In *23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, May 2001. IEEE.

[42] Hausi. A. Müller, Scott R. Tilley, and Kenny Wong. Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project. In *CASCON'93*, pages 217–226, October 1993.

[43] U. Nickel, J. Niere, J. Wadsack, and A. Zündorf. Roundtrip Engineering with FUJABA. In J. Ebert, B. Kullbach, and F. Lehner, editors, *Proceedings of the Second Workshop on Software-Reengineering (WSR)*, Bad Honnef, Germany, August 2000.

[44] Tobias Nipkow, Lawrence C. Paulson, and Merkus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer Verlag, 2002.

[45] Ken Pierce, Nico Plat, and Sune Wolff, editors. *Proceedings of the 8th Overture Workshop*, Technical Report CS-TR-1224. School of Computing Science, Newcastle University, 2010.

[46] Daniela Pohl. Specification Comprehension – Konzeptverwaltung am Beispiel zustandsbasierter Spezifikationen (in German). Master's thesis, University of Klagenfurt, Software Engineering and Soft Computing, Juli 2008.

[47] Daniela Pohl and Andreas Bollin. Database-Driven Concept Management – Lessons Learned from Using EJB Technologies. In *4th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 227–238. IEEE Computer Society, May 2009.

[48] Daniela Pohl and Andreas Bollin. Concept Management: Identification and Storage of Concepts in the Focus of Formal Z Specifications. *Communications in Computer and Information Science. Springer*, 69(II):248–261, 2010.

[49] Denys Poshyvanyk and Andrian Marcus. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC2007)*, pages 37–48, June 26–29 2007.

[50] Rational-XDE. IBM Rational XDE DeveloperWorks Home Page. www.ibm.com/ developer works/ rational/products/xde, Page last visited: April 2012.

[51] Philip E. Ross. The Exterminators. *IEEE Spectrum*, 42(9):36–41, September 2005.

[52] Ann E. Kelley Sobel and Michael R. Clarkson. Formal Methods Application: An Empirical Tale of Software Development. *IEEE Transaction on Software Engineering*, 28(3):308–320, March 2002.

[53] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 2nd edition, 1998.

[54] M.-A. D. Storey and H. A. Müller. Graph Layout Adjustment Strategies. In *Proceedings of Graph Drawing 1995*, Lecture Notes in Computer Science, pages 487–499, Passau, Germany, September 1995. Springer Verlag.

[55] Scott R. Tilley. Domain-Retargetable Reverse Engineering III: Layered Modeling. Technical report, Software Engineering Institute, Carnegie Mellon University, 1995.

[56] Mark Utting, Peter Robinson, and Ray Nickson. Ergo 6: a generic proof engine that uses Prolog proof technology. *LMS Journal of Computation and Mathematics*, 5:194–219, November 2002.

[57] Geoffry Norman Watson. *A Generic Proof Checker*. PhD thesis, The School of Computer Science. The University of Queensland, 2001.

[58] Jim Woodcock and Jim Davis. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, July 1996.

[59] Xinrong Xie, Denys Poshyvanyk, and Andrian Marcus. 3D Visualization for Concept Location in Source Code. In *Proceedings of 28th IEEE/ACM International Conference on Software Engineering (ICSE'06)*, pages 839–842, May 20–28 2006.

PART II

ASSESSMENT

CHAPTER 2

SLICE-BASED FORMAL SPECIFICATION MEASURES – MAPPING COUPLING AND COHESION MEASURES TO FORMAL Z

A. Bollin

In Cèsar Muñoz, editor, Proceedings of the Second NASA Formal Methods Symposium, NASA/CP-2010-216215, pages 24–34. NASA, Langley Research Center, April 2010.

Abstract

This paper demonstrates that existing slice-based measures can reasonably be mapped to the field of state-based specification languages. By making use of Z specifications this contribution renews the idea of slice-profiles and derives coupling and cohesion measures for them. The measures are then assessed by taking a critical look at their sensitiveness in respect to modifications on the specification source. The presented study shows that slice-based coupling and cohesion measures have the potential to be used as quality indicators for specifications as they reflect the changes in the structure of a specification as accustomed from their program-related pendants.

2.1 Introduction

In one of the rare articles concerning the relation between specifications and code, Samson, Nevill, and Dugard [13] demonstrate a strong quantitative correlation between size-based specification metrics and the related pendant of software code. Their assumption is that a meaningful set of (complexity and quality) measures could help in estimating product measures and development effort at a much earlier stage. Complexity can be described by

size-based attributes, but is it reasonable to measure the quality of a specification? This contribution takes a closer look at this problem.

Quality considerations are sophisticated. Besides the question of what a "good specification" looks like, quality-based measures (as in use with program code) are not so easily transformed to specifications. One reason is that such measures are usually based on control/data dependency considerations – concepts that are either not at all or only implicitly available. However, various authors demonstrated in [11, 4, 10, 17] that a reconstruction of the necessary dependencies ameliorates the situation and enables continuative techniques like slicing and chunking. And with that, slice-based measures (which are often taken as the basis for quality considerations) can be mapped to formal specifications, too. What would be the benefits of such measures?

As presumed by Samson et. al., with experiences from a large collection of specifications and implementations at hand, product and development estimates could be calculated at much earlier stages. But there is another benefit. When the measures are sensitive and react instantly to changes in the specifications, considerations, e.g. concerning deterioration effects, could be made, too.

The main objective of this contribution is now to investigate whether slice-based quality measures can reasonably be transformed to formal specifications. It does not invent new measures, but it maps the ideas behind the definitions of coupling and cohesion measures to the world of formal specification languages. Additionally, it looks for possible limitations. Based on Z [14], the mapping is described in some details and the outcome is assessed in respect to its expressiveness and sensitiveness.

This paper is structured as follows: Section 2.2 introduces specification slices and takes them as the basis for the slice-based measures mentioned above. Section 2.3 discusses the effects on the measures by making use of sample specifications, and Section 2.4 concludes the work with a short outlook.

2.2 Background

The motivation behind analyzing slice-based coupling and cohesion measures goes back to a paper of Meyers and Binkley [8]. In their empirical study they take a closer look at these measures and demonstrate that the values of coupling and cohesion can also be used for assessing deterioration effects. As formal specifications evolve, too, it would be interesting to see whether changes in the specification code show a similar behavior of these measures. As a necessary first step, a reasonable transformation of the original definitions of the measures to the world of formal specifications has to be found. This section demonstrates how this can be done for Z [14].

2.2.1 Slice-based Coupling and Cohesion

Coupling is a measure for the strength of inter-component connections, and cohesion is a measure for the mutual affinity of sub-components of a component. Within the range of this contribution we are interested in *how* these measures are calculated and *what* they indicate. As adumbrated in the introduction, a practical way in calculating coupling and cohesion measures is to make use of slices.

Weiser [15, 16] introduced five slice-based measures for cohesion: $Tightness$, $Overlap$, $Coverage$, $Parallelism$, and $Clustering$. Ott and Thuss [12] partly formalized these measures, and this contribution makes use of their formalization. *Coupling* was originally defined as the number of local information flow entering (fan-in) and leaving (fan-out) a procedure [7]. Harman et. al demonstrate in [6] that it can also be calculated via slicing. Furthermore, they show that the use of slices not only enables the detection of coupling, it can also be used to determine the "bandwidth" of the existing information flow. Their notion of information flow is also used in this contribution.

2.2.2 Specification Slices and Slice Profiles

For the calculation of coupling and cohesion measures, sets of slices and their intersections (comparable to the use of slice profiles in [12]) are needed. For state-based specifications the technique of slicing was introduced by Oda and Araki [11], informally redefined by Chang and Richardson [4], and then refined by Bollin [1]. His Java prototype has been extended in the recent years. It now supports slicing, chunking, and concept location of Z specifications [3]. The technical details of the identification of dependencies are not relevant within the scope of this paper, but the basic idea is quite simple:

First, the specification is dismantled into its basic elements called primes[1] by making use of the CZT parser [9]. The primes are mapped to a graph called SRN (for $Specification\ Relationship\ Net$). Then, by following the approach of Chang and Richardson and Bollin [4, 1] control and data dependencies are reconstructed (via a syntactical approximation to the semantical analysis). The SRN gets annotated by this dependency information, yielding an $Augmented\ SRN$ ($ASRN$ for short).

The $ASRN$ serves the same purpose as the system dependence graph used by the approaches described in [8, p.4]. Based on this data structure, slicing works as follows: a set of vertices (representing the point of interest) in the $ASRN$ is taken as starting point, and, by following the dependencies existing in the graph, further primes are aggregated, resulting in the designated specification slice. The transformation between a specification Ψ and its $ASRN$ is defined in a bijective manner. So, when talking about a specification it can be either the textual representation (consisting of a set of primes) or its $ASRN$ representation (consisting of vertices representing the primes).

Harman et. al [6] and Ott and Thuss [12] use different types of slices for their calculation of coupling and cohesion values. This situation is dealt with hereinafter by generating two variants of the static specification slices: for $coupling$ the slices are calculated by following the dependencies in a transitive backward manner, for the values of $cohesion$ the slices are calculated by combining the dependencies in a forward and backward manner. Specification slices and slice profiles (the collection of slices for a specific schema operation) are then defined as follows:

Definition 1 Static Specification Slice. *Let Ψ be a formal Z specification, ψ one schema out of Ψ, and V a set of primes v out of ψ. $SSlice_{fb}(\psi, V)$ is called static forward/backward specification slice of ψ for primes V. It is calculated by generating a transitive forward and backward slice with V as the starting point of interest. When the slice is generated in a transitive and backward manner, it is called static backward slice $SSlice_b(\psi, V)$.*

[1] Basically, primes are the predicates of the specification and are later represented as vertices in an augmented graph. When they represent predicates of the precondition of a schema they are called precondition primes, and when they form predicates that represent after-states they are called postcondition primes.

Definition 2 Slice Profile, Slice Intersection, Slice Union. *Let Ψ be a formal Z specification, ψ one schema out of Ψ, and V the set of primes v representing all postcondition primes in ψ. The set of all possible static specification slices ($SSlice_{fb}(\psi, \{v\})$ or $SSlice_{b}(\psi, \{v\})$, with $v \in V$) is called Slice Profile ($SP(\psi)$). The intersection of the slices in $SP(\psi)$ is called Slice Intersection ($SP_{int}(\psi)$). The union of all slices in $SP(\psi)$ is called Slice Union ($SU(\psi)$).*

2.2.3 Cohesion

With the introduction of slice profiles it is possible to provide the definitions of cohesion measures (as introduced in the work of Ott and Thuss [12]). The values for cohesion are calculated only for a given schema. As slices and slice profiles might contain primes from other schemata (due to inter-schema dependencies), the following definitions restrict the set of primes in the slice profile to the schema.

Definition 3 Tightness. *Let Ψ be a formal Z specification, ψ one schema out of Ψ, $SP(\psi)$ its slice profile, and $SP_{int}(\psi)$ its slice intersection. Then Tightness $\tau(\psi)$ is the ratio of the size of the slice intersection to the size of ψ. It is defined as follows:*

$$\tau(\psi) = \frac{\mid SP_{int}(\psi) \cap \psi \mid}{\mid \psi \mid}$$

Definition 4 MinCoverage, Coverage, MaxCoverage. *Let Ψ be a formal Z specification, ψ one schema out of Ψ, and $SP(\psi)$ its slice profile containing n slices. MinCoverage $Cov_{min}(\psi)$ expresses the ratio between the smallest slice SP_{i-min} in $SP(\psi)$ and the number of predicate vertices in ψ. Coverage $Cov(\psi)$ relates the sizes of all possible specification slices SP_i ($SP_i \in SP(\psi)$) to the size of ψ. MaxCoverage $Cov_{max}(\psi)$ expresses the ratio of the largest slice SP_{i-max} in the slice profile $SP(\psi)$ and the number of vertices in ψ. They are defined as follows:*

$$Cov_{min}(\psi) = \frac{1}{\mid \psi \mid} \mid SP_{i-min} \cap \psi \mid \qquad Cov(\psi) = \frac{1}{n} \sum_{i=1}^{n} \frac{\mid SP_i \cap \psi \mid}{\mid \psi \mid}$$

$$Cov_{max}(\psi) = \frac{1}{\mid \psi \mid} \mid SP_{i-max} \cap \psi \mid$$

Definition 5 Overlap. *Let Ψ be a formal Z specification, ψ one schema out of Ψ, $SP(\psi)$ its slice profile containing n slices, and SP_{int} its slice intersection. Then Overlap $O(\psi)$ measures how many primes are common to all possible specification slices SP_i ($SP_i \in SP(\psi)$). It is defined as follows:*

$$O(\psi) = \frac{1}{n} \sum_{i=1}^{n} \frac{\mid SP_{int} \cap \psi \mid}{\mid SP_i \cap \psi \mid}$$

Tightness measures the number of primes that are common to every slice. The definition is based on the size[2] of the slice intersection. Coverage is split into three different measures: *Minimum Coverage* looks at the size of the smallest slice and relates it to the size of the specification, *Coverage* looks at the size of the slices, but it takes all slices and compares them to the size of the specification, and *Maximum Coverage* looks at the size of the largest slice and relates it to the size of the specification. Finally, *Overlap* looks at the slice intersection and determines how many primes are common to all slices.

[2]Please note that within the context of all definitions *size* counts the number of primes in the $ASRN$.

2.2.4 Coupling

The calculation of coupling follows the definitions to be found in [6]. First, Inter-Schema Flow F is specified. It describes how many primes of the slices in the slice union are outside of the schema. Inter-Schema Coupling then computes the normalized ratio of this flow in both directions.

Definition 6 Inter-schema Flow and Coupling. *Let Ψ be a formal Z specification and ψ_s and ψ_d two schemata out of Ψ. Inter-Schema Flow between the two schemata $F(\psi_s, \psi_d)$ is the ratio of the primes of $SU(\psi_d)$ that are in ψ_s and that of the size of ψ_s. Inter-Schema Coupling between the two schemata $C(\psi_s, \psi_d)$ measures the Inter-Schema Flow in both directions. They are defined as follows:*

$$F(\psi_s, \psi_d) = \frac{\mid SU(\psi_d) \cap \psi_s \mid}{\mid \psi_s \mid}$$

$$C(\psi_s, \psi_d) = \frac{F(\psi_s, \psi_d) \times \mid \psi_s \mid \; + \; F(\psi_d, \psi_s) \times \mid \psi_d \mid}{\mid \psi_s \mid + \mid \psi_d \mid}$$

Schema coupling is calculated by considering the Inter-Schema Coupling values to all other schemata.

Definition 7 Schema Coupling. *Let Ψ be a formal Z specification and ψ_i one schema in Ψ. Then Schema Coupling $\chi(\psi_i)$ is the weighted measures of the Inter-Schema Coupling of ψ_i to all other n schemata ψ_j in $\Psi \setminus \psi_i$. It is defined as follows:*

$$\chi(\psi_i) = \frac{\sum_{j=1}^{n} C(\psi_i, \psi_j) \times \mid \psi_j \mid}{\sum_{j=1}^{n} \mid \psi_j \mid}$$

With the measures in this section it is possible to assign attributes to a formal Z specification. However, with the mapping a connection to quality has been so far not empirically justified. On the other hand, the slice-based measures have carefully been transformed to Z. There is a chance that, when observing *changes* of these values for a given specification, one might defer useful properties.

2.3 Sensitivity of Slice-based Measures

By following the strategy that Thuss and Ott [12] used for their validations, we now investigate the *sensitivity* of the measures with respect to *representative changes* of the specifications. The advantage is that for such a study only small-sized sample specifications are necessary to explore the effects.

2.3.1 Sensitivity of Cohesion

The first objective is to determine whether the transformed measures for cohesion are sensitive to changes in the internal structure of the specification. The following operations are considered:

O1 Adding a precondition-prime. This means that this prime ”controls” the evaluation of the other primes in the schema. With it, the internal semantic connections are

$SP(\psi)$			Specifications	Measures
		—	*Test1* $n, n' : \mathbb{N}$ $m, m' : \mathbb{N}$ $n' = n + 1$	$\#SP_{int}(\psi) = 1$ $\tau(\psi) = 1.00$ $Cov_{min}(\psi) = 1.00$ $Cov(\psi) = 1.00$ $Cov_{max}(\psi) = 1.00$ $O(\psi) = 1.00$
	—	—	*Test2* $n, n' : \mathbb{N}$ $m, m' : \mathbb{N}$ $n' = n + 1$ $m' = m + 1$	$\#SP_{int}(\psi) = 0$ $\tau(\psi) = 0.00$ $Cov_{min}(\psi) = 0.50$ $Cov(\psi) = 0.50$ $Cov_{max}(\psi) = 0.50$ $O(\psi) = 0.00$
	— —	— —	*Test3* $n, n' : \mathbb{N}$ $m, m' : \mathbb{N}$ $delta? : \mathbb{N}$ $delta? > 0$ $n' = n + delta?$ $m' = m - delta?$	$\#SP_{int}(\psi) = 1$ $\tau(\psi) = 0.33$ $Cov_{min}(\psi) = 0.67$ $Cov(\psi) = 0.67$ $Cov_{max}(\psi) = 0.67$ $O(\psi) = 0.50$
	— — —	— — —	*Test4* $n, n' : \mathbb{N}$ $m, m' : \mathbb{N}$ $delta? : \mathbb{N}$ $set? : \mathbb{P}\mathbb{N}$ $delta? > 0$ $set? \neq \varnothing$ $n' = n + delta?$ $m' = m - delta?$	$\#SP_{int}(\psi) = 2$ $\tau(\psi) = 0.50$ $Cov_{min}(\psi) = 0.75$ $Cov(\psi) = 0.75$ $Cov_{max}(\psi) = 0.75$ $O(\psi) = 0.67$

$SP(\psi)$			Specifications	Measures
— — —	— — — 	— — — 	*Test5* $n, n' : \mathbb{N}$ $m, m' : \mathbb{N}$ $delta? : \mathbb{N}$ $set? : \mathbb{P}\mathbb{N}$ $p, p' : \mathbb{N}$ $delta? > 0$ $set? \neq \varnothing$ $n' = n + delta?$ $m' = m - delta?$ $p' = p + delta?$	$\#SP_{int}(\psi) = 2$ $\tau(\psi) = 0.40$ $Cov_{min}(\psi) = 0.60$ $Cov(\psi) = 0.60$ $Cov_{max}(\psi) = 0.60$ $O(\psi) = 0.33$
— — — —	— — — 	— — — —	*Test6* $n, n' : \mathbb{N}$ $m, m' : \mathbb{N}$ $delta? : \mathbb{N}$ $set? : \mathbb{P}\mathbb{N}$ $p, p' : \mathbb{N}$ $delta? > 0$ $set? \neq \varnothing$ $n' = n + delta?$ $m' = m - delta?$ $p' = p + n$	$\#SP_{int}(\psi) = 2$ $\tau(\psi) = 0.40$ $Cov_{min}(\psi) = 0.60$ $Cov(\psi) = 0.73$ $Cov_{max}(\psi) = 0.80$ $O(\psi) = 0.56$
	— — — —	— — — —	*Test7* $n, n' : \mathbb{N}$ $m, m' : \mathbb{N}$ $delta? : \mathbb{N}$ $set? : \mathbb{P}\mathbb{N}$ $delta? > 0$ $set? \neq \varnothing$ $n' = n + delta?$ $m' = m - n$	$\#SP_{int}(\psi) = 4$ $\tau(\psi) = 1.00$ $Cov_{min}(\psi) = 1.00$ $Cov(\psi) = 1.00$ $Cov_{max}(\psi) = 1.00$ $O(\psi) = 1.00$

Figure 2.1 Z specifications of raising sizes. On the left side of the table the slices (and thus the slice-profile) are visualized, on the right side the values for cohesion are presented.

extended. Mapped to a potential implementation, this could mean that an if-clause is added to the code, enveloping all other statements in the method. This operation should slightly increase coverage.

O2 Adding a prime that specifies an after-state and that is not related to all the other predicates in the schema. In this case the predicate introduces new "trains of thought". Mapped to a subsequent implementation, this could mean that a new output- or state-relevant statement (not or only fractionally related to the other statements) is added. With it, a new slice is added to the slice-profile. The slice intersection is very likely smaller than before, thus reducing the values for cohesion.

O3 Adding a prime that specifies an after-state and that is furthermore related to all other primes in the schema. In this case the predicate extends existing "trains of thought" (as there are references to all existing ones). Mapped to a possible implementation, it is very likely that a new output- or state-relevant statement, related to all other statements, is added. If at all, this increases the set of intersection slices. And with that, it also raises the values of coupling.

Based on the assumption that schema operations are often mapped to methods (or procedures) as described in operations $O1$ to $O3$, the following working hypothesis can be posted:

Hypothesis 1 *A structural change of type $O1$, $O2$ or $O3$ in a schema operation influences the values for cohesion. Adding a predicate prime to the schema according to operations $O1$ or $O3$ increases the values (or leaves them unchanged), adding a prime according to operation $O2$ decreases the values (or leaves them unchanged). Reversing the operations also reverses the effect on the measures.*

There are situations where, due to a large number of dependencies, a method or a schema operation already has reached the maximum values for cohesion. These special cases are the reason why the values might also be unchanged (and Sec. 2.3.3 reconsiders this issue in more details).

Hypothesis 1 is now checked by using small sample schema operations (called $Test1$ to $Test7$ in Fig. 2.1). At first let us start with a simple Z schema operation called $Test1$. It contains a prime that increases the value of n by one. As there is only one slice, the slice intersection only contains one element. The values of cohesion are all 1. Then another prime ($m' = m + 1$, prescribing an after-state) is added to the schema (which is an operation of class $O2$), yielding operation $Test2$. With this new prime a new "functionality" has been introduced to the schema. The values for cohesion are reduced as the slice intersection is empty. Tightness and Overlap are zero, the rest of the values are equal to $\frac{1}{2}$. Then, in $Test3$, the prime $delta? > 0$ is added to the schema. This prime is a precondition prime and thus the operation belongs to class $O1$. With this, all the values of cohesion increase. As the slice intersection contains one prime only ($delta? > 0$), its size is $\#SP_{int}(\psi) = 1$. With that, the values for cohesion result in: $\tau = \frac{1}{3}$, $Cov_{min} = \frac{1}{3} \times 2$, $Cov = \frac{1}{2} \times (\frac{2}{3} + \frac{2}{3})$, $Cov_{max} = \frac{1}{3} \times 2$, and $O = \frac{1}{2} \times (\frac{1}{2} + \frac{1}{2})$.

$Test4$ adds another precondition prime $set? \neq \varnothing$ to the schema (operation $O1$). This yields an increase in the values of cohesion. The reason is the increase in size of the slice intersection. $Test5$ adds another prime containing an after state identifier to the schema operation. The prime $p' = p + delta?$ is not related to the other primes prescribing after-states, so this change is an operation of class $O3$. The size of the slice intersection stays the same, only the size of the schema increases. As a result the values for cohesion decrease.

Now let us take a look at situations when predicates that are partly related to existing predicates are added to the schema. $Test6$ is an extension of $Test4$, but this time the new prime $p' = p + n$ uses the identifier n which is also defined by the postcondition prime $n' = n + delta?$. On the other hand it does not refer to the third postcondition prime $m' = m - delta?$, and so the operation belongs to class $O2$. The values for cohesion consequently decrease. On contrary, $Test7$ is a modification of $Test3$. The prime $m' = m - n$ is added, and so it is related to all other postcondition primes in the schema. With this operation the values for cohesion increase, again.

Due to reasons of space the example schemata above only contain simple predicates. But they are sufficient to demonstrate the influence of structural changes. In Z there are several schema operators that complicate the situations, but by further analyzing the formulas of the measures one observes the following behavior:

- Cohesion will increase when (a) at least one postcondition exists and a precondition prime is added, (b) a postcondition prime that is related to some, but not to all, other postcondition primes in the schema is added, (c) a postcondition prime that is not related to the other postcondition primes is removed.

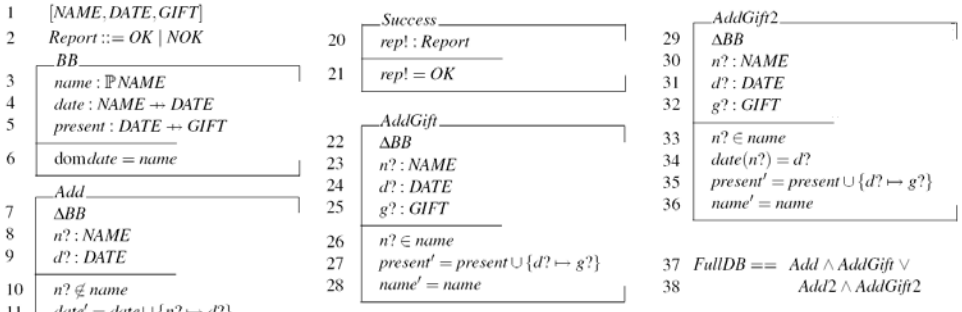
- Cohesion stays the same when (a) a postcondition prime that is related to all other existing postcondition primes is added or removed and the other existing primes are already fully inter-related, (b) a pre- or postcondition prime is added and there is no postcondition prime.

- Cohesion will decrease when (a) a postcondition prime that is not related to the other postcondition primes is added, (b) a precondition prime is removed and there is at least one postcondition prime, (c) a postcondition prime that is related to the other postcondition primes is removed.

The values for a derived implementation would very likely react to these changes in the same way, so the sample specifications and the analysis of the formulas seems to confirm our working hypothesis. The measures are sensitive to changes in the underlying specification, and the observed changes are in such a way that an increase in internal connectivity also leads to an increase of the values of the measures. Conversely, a decrease in connectivity also leads to a decrease of the values of cohesion.

2.3.2 Sensitivity of Coupling

The next step is the evaluation of coupling. As mentioned above, specification coupling measures the mutual interdependence between different schemata. According to Harman et. al [6] it is an advantage to use slice-based measures as they measure the ”bandwidth” of the information flow. In our case, this flow is defined as the relative amount of the size of a set of slices of one schema that lies inside another schema. This flow depends upon control- and data-relationships, so an increase in the number of these dependencies should increase the value of coupling. Reducing the number of relations should decrease the value of coupling. There are no differences between the definitions of the measure, be it for traditional programs or be it for formal specifications.



Schema	Coupling	τ	Cov.	O
(S1) *BB*	0.775	1.000	1.000	1.000
(S2) *Add*	**0.724**	0.750	0.875	0.958
(S3) *Add2*	**0.737**	0.800	0.867	0.972
(S4) *Success*	0.000	1.000	1.000	1.000
(S5) *AddGift*	**0.724**	0.750	0.875	0.958
(S6) *AddGift2*	**0.737**	0.800	0.900	0.958

Figure 2.2 Z example for analyzing the effect of slice-based coupling and values for coupling and cohesion for the six Z schemata (omitting the *FullDB* operation schema).

The next hypothesis focuses on the sensitiveness to structural changes within a formal Z specification. Especially, an increase (or decrease) in inter-schema relations should be reflected correctly[3].

Hypothesis 2 *A change in the number of relations between two schemata in a formal specification generally leads to a change in the value of coupling. Adding a predicate prime to one schema that introduces new dependencies to primes in the other schema increases the value of coupling (or leaves it unchanged). Reversing the change also reverses the effect on the measure.*

For our considerations we now make use of a small specification (see Fig. 2.2) which is an extension (and intentionally unfinished version) of the birthday-book specification out of Spivey [14, pp.3–6].

The specification consists of one state space called BB (for $Birthday\ Book$) which stores the names of friends, their birthday dates, *and* a small gift. Consequently, there are two operations (Add and $AddGift$) for adding them to the database. The operations are not total, but are sufficient for our examinations. In order to analyze the effect of added pre- and postcondition primes, these operations are available in two versions. Finally, there is an operation called $Success$ which should (later on) indicate the success of an operation. However, at the moment this operation is not related to any of the other operations.

The values of schema coupling are summarized in Fig. 2.3. As expected, the $Success$ operation has a value of coupling equal to zero. There are no connections to the state space BB and also no connections to the other operation schemata. On the other hand, the values for the other operations differ (though their syntactical composition is more or less

[3]Again, there are situations where, due to a high number of dependencies, the value of coupling might stay unchanged.

F	$(S1)$	$(S2)$	$(S3)$	$(S4)$	$(S5)$	$(S6)$
$(S1)$	1.000	**1.000**	1.000	0.000	1.000	1.000
$(S2)$	**0.750**	1.000	0.750	0.000	0.750	0.750
$(S3)$	0.800	0.800	1.000	0.000	0.800	0.800
$(S4)$	0.000	0.000	0.000	1.000	0.000	0.000
$(S5)$	0.750	0.750	0.750	0.000	1.000	0.750
$(S6)$	0.800	0.800	0.800	0.000	0.800	1.000

C	$(S1)$	$(S2)$	$(S3)$	$(S4)$	$(S5)$	$(S6)$
$(S1)$	1.000	**0.800**	**0.833**	0.000	0.800	0.833
$(S2)$	0.800	1.000	0.778	0.000	**0.750**	**0.778**
$(S3)$	0.833	0.778	1.000	0.000	0.778	0.800
$(S4)$	0.000	0.000	0.000	1.000	0.000	0.000
$(S5)$	**0.800**	0.750	0.778	0.000	1.000	0.778
$(S6)$	**0.833**	0.778	0.818	0.000	**0.778**	1.000

Figure 2.3 Values for Inter-Schema Flow $F(\psi_1, \psi_2)$ and Inter-Schema Coupling $C(\psi_1, \psi_2)$. The abbreviations $S1$ to $S6$ refer to the schema names mentioned in Fig. 2.2.

equivalent). With $n = 6$ operations there are 15 different combinations and thus possibly 15 values for Inter-Schema Coupling. Within the scope of this contribution we will focus on three combinations that are of most interest in this schema constellation.

As a first example we look at the operations Add and $Add2$. The difference between them is made up by just one prime, namely $name' = name \cup \{n?\}$ at line 18. In fact, in the context of the specification this postcondition prime is redundant (as the state invariant at line 6 would guarantee that the added name is also in the set of names). But syntactically this prime is sufficient to increase the set of dependencies. Both operations include the state-space, which means that there is a relation between the postcondition primes and the invariant. This introduces a new "flow of control", which increases the bandwidth and thus the value for coupling (from 0.724 to 0.737 in Fig. 2.2).

Fig. 2.3 presents the values for Inter-Schema Flow and Coupling, and they make this difference more visible. Inter-Schema Flow $F(BB, Add)$ is $\frac{|SU(Add) \cap BB|}{|BB|}$, which is 1 (so the slices of Add(S2) cover 100% of the state space BB(S1)). The value of $F(Add, BB)$ is $\frac{|SU(BB) \cap Add|}{|Add|}$, where $SU(BB) \cap Add$ covers the primes at lines $\{6, 10, 11\}$ (due to data dependency between line 6 and line 11). With this, the Inter-Schema Flow is $\frac{3}{4}$. (Please note that due to the schema inclusion the Add schema consists of 4 predicates!) Now, looking at Inter-Schema Coupling, the value is $\frac{1 \times 1 + 3/4 \times 4}{1+4}$, which is 0.8. Similarly, one can calculate the value for the coupling between $BB(S1)$ and $Add2(S3)$, which is 0.833. The new dependency between the invariant at line 6 and line 18 led to the situation that the slice contains one more prime. For similar situations we might infer that the introduction of postcondition primes will (very likely) raise the value of coupling.

Another situation occurs when looking at the operation schemas $AddGift$ and $AddGift2$. In relation to the state schema the second variant of the operation contains an additional prime at line 35. However, the point of departure is not the same. In this case the prime is a precondition prime that does not influence any primes outside the schema – at first sight. On closer examination it is clear that the postcondition primes in this schema are control dependent upon this prime, and a slice "reaching" the schema operation will have to include this prime, too. It grows in size, meaning that more "bandwidth" is spent on it. In similar situation we can infer that the value of coupling will also increase as the value for the Inter-Schema Flow increases from $\frac{n}{m}$ to $\frac{n+1}{m+1}$.

The above specification does not show that the value for coupling can also decrease. This is the case when we introduce a postcondition prime that is not related to the primes in the other schema(ta). Then, in case of a state schema, there is no data-dependency between the primes. And in the case of another operation schema there is neither control

nor data-dependency. As the size of the schema increases, Inter-Schema Flow decreases from $\frac{n}{m}$ to $\frac{n}{m+1}$.

On the syntactical level there is no difference between Add and $AddGift$. Both consist of a precondition prime, three postcondition primes, and include the state. This equivalence can be seen in Fig. 2.2 as the values for cohesion are the same. But it gets interesting when comparing them to $AddGift2$. The value for Inter-Schema Coupling between Add ($S2$) and $AddGift$ ($S5$) is 0.750, whereas the value for Add ($S2$) and $AddGift2$ ($S6$) is 0.778. So, there is a slightly higher value of coupling between Add and $AddGift2$. The reason for this is a semantic difference: the data relationship between the lines 11 and 34. This simple example demonstrates that the idea of the "bandwidth" is quite applicable in this situation.

Though the above example is simple, it is able to demonstrate the effects on the value for coupling in the case of structural changes of schema operations. The second working hypothesis seems to be confirmed, at least from the analytical part of view.

2.3.3 Limitations

Though the above hypotheses seem to be confirmed, there are limitations. More precisely, the problems are that (a) the specifications might be too dense, (b) only part of the "bandwidth" is regarded, and (c) the dependency reconstruction does not work correctly. What seems to corrupt the measures is in fact not a real problem.

In [2] the effect and efficiency of slicing has been investigated, and it turned out that slicing has disadvantages when the specifications are too dense. In about 60-70% of all cases slicing led to a reduction in the size of the specification, which also means that in some situations there has been no reduction at all. The slices then contained all other primes – indicating that nearly every prime is related to all other primes. However, this effect decreases on average with raising sizes of the specifications (our experience relies on more than 45.000 lines of specification text), and it is only a problem with text-book examples that consist of a view schema operations only.

The next concern is that coupling is not sensitive to changes that lead to an increase in the number of relations between the same primes. Irrespective the number of dependencies between two primes, only the occurrence of the primes is counted by the size-operator. Bandwidth does not increase then. The presented notion of coupling works well on a syntactical level, but not necessarily as expected on a semantic level. The last measure (comparing $AddGift$ and $AddGift2$) was sensitive because one prime has (intentionally) been omitted from both schemata: normally, an author of these operations would have added the line $date' = date$ as predicates to the schemata. This would have introduced data-dependencies from both schemata to the Add schema, and there would have been no difference in Inter-Schema Coupling anymore. In fact, this can not be seem as a real problem, it is as coupling is defined. Nevertheless, one has to keep in mind that there might be more dependencies as expected.

Finally, slicing only works fine when the specifications are "well-formed". This means that they consist of separable pre- and postconditions primes. When such a separation is not possible, then the outcome is vitiated. Diller mentions in [5, p.165] that in most cases this separation can be done (which means that a syntactical approximation to the semantic analysis is possible), but this does not prevent from cases where pre- and postconditions are interwoven and not separable.

2.4 Conclusion and Outlook

This contribution introduces the notion of specification slice-profiles that are then used for the calculation of slice-based specification measures. The way of calculating these measures for Z (namely coupling and cohesion) is new and it is based on the use of (reconstructed) control and data dependency information. The objective of this work is to investigate if such a mapping is meaningful. For this, the contribution takes a set of small specifications as a basis, and the sensitivity of the measures is then analyzed by changing internal and external properties of the specifications.

The evaluation shows that the measures reflect the changes in the structure of the specification as expected. Especially the values for cohesion seem to be a good indicator for changes in internal properties. Coupling is, due to the use of unions of slices a bit less sensitive, but it also reacts when there are dramatic structural changes. All in all, the measures proof useful.

The understanding of the *behavior* of the measures was a first, necessary step. The next steps will be to include different "real-life" specifications and to perform an empirical study that demonstrates that the measures are not just proxies for other, eventually size- based, measures. In case of confirming their unique features again, this could be a step towards taking specifications as quality indicators quite at the beginning of the SW-development cycle.

REFERENCES

[1] Andreas Bollin. *Specification Comprehension – Reducing the Complexity of Specifications*. PhD thesis, Universität Klagenfurt, April 2004.

[2] Andreas Bollin. The Efficiency of Specification Fragments. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering*, 2004.

[3] Andreas Bollin. Concept Location in Formal Specifications. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(2):77–104, March/April 2008.

[4] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[5] Antoni Diller. *Z – An Introduction to Formal Methods*. John Wiley and Sons, 2nd edition, 1999.

[6] Mark Harman, Margaret Okulawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of coupling. In *Proceedings of the IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution. Boston, Massachusetts*, pages 28–32, 1997.

[7] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

[8] Timothy M. Meyers and David Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(1), December 2009.

[9] Tim Miller, Leo Freitas, Petra Malik, and Mark Utting. CZT Support for Z Extensions. In *Proc. 5th International Conference on Integrated Formal Methods (IFM 2005)*, pages 227 – 245. Springer, 2005.

[10] Roland T. Mittermeir and Andreas Bollin. Demand-driven Specification Partitioning. *Lecture Notes in Computer Science*, 2789:241–253, 2003.

[11] Tomohiro Oda and Keijiri Araki. Specification slicing in a formal methods software development. In 17^{th} *Annual International Computer Software and Applications Conference*, IEEE Computer Socienty Press, pages 313–319, November 1993.

[12] Linda M. Ott and Jeffrey J. Thuss. Slice based metrics for estimating cohesion. In *In Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81. IEEE Computer Society Press, 1993.

[13] W.B. Samson, D.G. Nevill, and P.I. Dugard. Predictive software metrics based on a formal specification. In *Information and Software Technology*, volume 29 of *5*, pages 242–248, June 1987.

[14] J.M Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 2^{nd} edition, 1992.

[15] Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.

[16] Mark Weiser. Program slicing. In *Proceedings of the* 5^{th} *International Conference on Software Engineering*, pages 439–449. IEEE, 1982.

[17] Fangjun Wu and Tong Yi. Slicing Z Specifications. *SIGPLAN Not.*, 39(8):39–48, 2004.

CHAPTER 3

IS THERE EVOLUTION BEFORE BIRTH? – DETERIORATION EFFECTS OF FORMAL Z SPECIFICATIONS

A. Bollin

Lecture Notes in Computer Science, 6991 (Formal Methods and Software Engineering): 66–81. 2011.

Abstract

Formal specifications are not an exception for aging. Furthermore, they stay valid resources only in the case when they have been kept up to date during all evolutionary changes taking place. As specifications are then not just written once, an interesting aspect is whether they do also deteriorate or not. In order to answer this question, this paper addresses the issues on various kinds of changes in the development of formal specifications and how they could be measured. For this, a set of semantic-based measures is introduced and then used in a longitudinal study, assessing the specification of the Web-Service Definition Language. By analyzing all 139 different revisions of it, it is shown that specifications can deteriorate and that it takes effort to keep them constantly at high quality. The results yield in a refined model of software evolution exemplifying these recurring changes.

3.1 Introduction

Would you step into a house when there is a sign saying "Enter at your own risk"? I assume not, at least if it is not unavoidable. Well, the situation is quite comparable to a lot of software systems around. Our standard software comes with license agreements stating

that the author(s) of the software is (are) not responsible for any damage it might cause, and the same holds for a lot of our hardware drivers and many other applications around. Basically, we use them at our own risk.

I always ask myself: "Would it not be great to buy (and also to use) software that comes with a certificate of guarantee instead of an inept license agreement?" Of course, it would and it is possible as some companies demonstrate. It is the place where formal methods can add value to the development process. They enable refinement steps and bring in the advantages of assurance and reliable documentation.

The argument of quality is not just an academic one. Formal methods can be used in practice as companies using a formal software development process demonstrate [23]. Formal modeling is also not as inflexible as one might believe. Changing requirements and a growing demand in software involve more flexible processes and it is good to see that a combination of formal methods and the world of agile software development is possible [2]. This enables the necessary shorter development cycles, but, and this is the key issue, it also means to start thinking about evolution right from the beginning.

The questions that arise are simple: (a) Do our formal specifications really change or evolve, and (b) if this is the case, can we detect or even measure these changes? The objective of the paper is to answer these two questions. In a first step it demonstrates that formal specifications are not an exception for aging. Section 2 tries to make developers more receptive to this topic. And in the second step it demonstrates that there might be deterioration effects when formal specifications are undergoing constant changes. For this, Section 3 briefly introduces a set of measures that are suitable for assessing Z specifications, and Section 4 takes a closer look at 139 revisions of a large Z specification. Due to the lessons learned, a refined model of software evolution is suggested in Section 5. Finally, Section 6 summarizes the findings and again argues for a careful attention of the refined model of (specification) evolution.

3.2 Perfection or Decay

A formal specification describes what a system should do and as such it can be used to argue about the correctness of a candidate system. But a specification is not per se a "correct" mapping of the requirements. It needs time to create a first, useful version and, as there are affinities with traditional software development, this section starts with the model of software evolution. This model is then the basis for a – necessary – refinement, as is shown later in Section 5.

3.2.1 Back to the Roots

Let us start again with the analogy above: Why does one enter a house even without bothering about its safety? The answer is simple: normally, one trusts in the original design, the statics, the teams that built it and the officials that did the final checks. The trust stays the same when the house is going to be renovated, when the interior changes and when some walls are broken down (or new ones are erected). One naturally assumes that the old plans have been investigated and that structural engineers took a look at it. The same holds for our software systems. There is an overall design, there are teams that build and modify it and there are teams that test it before being sold. We trust in their professionalism. A change in requirements and needs then leads to a change in this software – it is undergoing a "renovation" process that we might call software evolution.

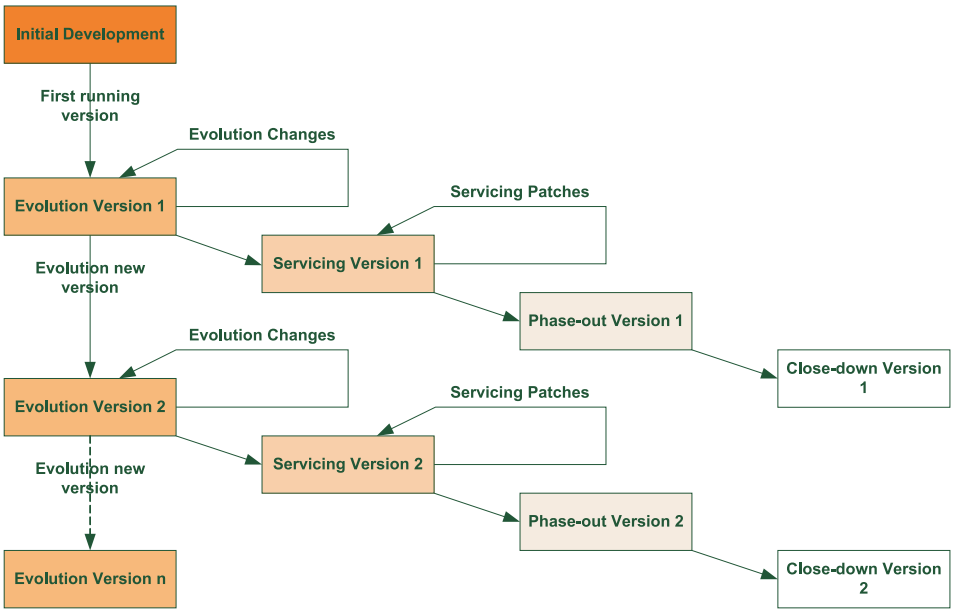


Figure 3.1 The versioned staged model of Bennett and Rajlich [1]. Starting with the initial development and the first running version evolution is about to begin. The goal of the evolution phase is to adapt the software to ever changing user requirements or changes in the operating environment. When substantial changes are not possible anymore (at least not without damages to the system), then the servicing phase starts. Only small tactical changes are applied. When no more servicing is done, then the phase-out starts. Finally, with the close-down phase, the users are directed towards a new version.

Bennet and Rajlich [1] introduced a staged model to describe this process in more details (see Fig. 3.1). Starting with the initial development and the first running version, evolutionary changes happen, leading to servicing phases and then, finally, to the phase-out and close-down versions of the software. In their article the authors also point out the important role of software change for both, the evolution and servicing phases. In fact, effort is necessary at every step of the phase to keep up with the quality standards and for keeping alive our trust in it.

Taking a closer look at our analogy of building/reconstructing a house it can be observed that there is also a chain (or network) of trust. The visitor (or owner) of the house counts on the construction company, they by themselves trust in the quality of the building materials they use, and the team that builds the house trusts in the architects (just to mention some of the links). When our evolving software is undergoing changes, then there is a similar chain of trust and dependencies.

This is now the place where formal methods come into play. To keep the trust, a change in the software has to be preceded by changes in the design documents and with it a change in the software specifications. One can also put it the other way round: when the architect does not update his or her plans, then future renovations are (hopefully) impeded.

3.2.2 The Role of Formal Design

Writing down requirements in a keen way is necessary, and the use of formality is not new in this respect. In their article Black et.al point out that formal methods have already been used by Ada Lovelace's and Charles Babbage's work on the analytical engine when they verified the formulas [2]. Since then several success stories of the use of formal methods have been published [8, 28, 23, 12, 11]. However, traditional formal design is commonly seen as something that is just happening at the beginning, and most of us are tempted to think about the development of just *one* formal model.

As the analogy above demonstrates, this viewpoint is probably not correct. When drawing up a plan, an architect does not only draw a single version. He or she starts with a first version, modifies it, and plays around with it. Several versions are assessed and, finally, discussed with the customer. The plans are, in addition to that, revised when the building is changed later on. The same holds for formal specifications. Their advantages not only lie in verification and validation considerations. They form the basis for incrementally pinning down the most important requirements (and eventually ongoing refinement steps). Only when kept up to date during evolutionary changes, they act as valid sources for comprehension activities. So, our formal specifications are (and should be) constantly changing during the software development phases. To think in terms of "write-once languages" (as already addressed in [16, p.243]) is for sure not appropriate.

During the last three decades there have been several advances in the field of formal software development. Specification languages were standardized and then also adapted to the world of object oriented programming. By the time new languages arise. Alloy is such an example that also allows for simulation and graphical feedback [13]. However, the main focus of the tools and languages around rests on support for writing the one and correct model (and on proving properties of it). Contrarily, in our working group we have been focusing on servicing and comprehension aspects instead [21, 17], and in the last years these efforts led to a concept location model and a tool for visualization, slicing and clustering of Z specifications [4]. The resulting tools and techniques will now be used in order to find out whether (and to which extent) formal specifications do change during evolutionary activities.

3.3 On the Search for Measures

Formal specifications (like program code) might age during modifications and it needs effort to antagonize it. The effects of a modification should be measured in order to steer the course of change. This means to assess the specification (among other documents) at every refinement step and to consider the effect on various parameters of the specification. Looking at size-based measures only (which can be calculated easily) is for our objectives not enough. When talking about various aspects of deterioration we are more interested in measuring effects on the specifications' *quality*!

3.3.1 Specification Measures

The majority of specification metrics used in projects belongs to the class of size or quantity based measures. Most popular is counting lines of specification text, which, apart from looking at the time needed to write the specification, was also used as *the* basis to monitor

the often-cited CICS/ESA project of IBM [9]. Counting specific specification elements is possible, too. Vinter et. al propose to count the type and number of logical constructs in Z specifications [25]. By a small case-study they demonstrate that these measures might correlate with the complexity of the specification. However, up to now a quantitative assessment of the approach is missing. Nogueira et. al suggest to use two measures expressing the complexity of each operator in the system and to calculate them by counting input and output data related to the operators [18]. Their experiences are based on a small case-study using $HOPE$ as a specification language and Modula-2 for its implementation. Alternatively, Samson et. al suggest to count the number of equations in a module or per operation [24]. They observed that the number of equations required to define an operator is frequently equal to the cyclomatic complexity of code based on the specification.

Complexity considerations are relevant, but within the scope of this work quality measures are needed. Due to the declarative nature of the formal specifications under investigation, such measures (usually based on control- and data-flow) are, unfortunately, rare. The above mentioned approaches of Samson et. al or Nogueira et. al can be seen, if at all, just as possible approximations to quality considerations. But, there is one approach that could be used as a starting point. By looking at (and analyzing) the references to state variables in operations, Carrington et. al try to identify states of a module and to relate them to the top-level modular structure of an implementation [6]. With this, they are introducing the notion of cohesion within a module. They do not relate it to the quality of a specification, though, but the next section demonstrates that not so much is missing.

3.3.2 Slice-based Coupling and Cohesion Measures

As mentioned in the previous section, it is hard to find suitable quality measures for formal specifications. However, for programming languages there are several approaches around. Recently, Meyers and Binkley published an empirical study demonstrating the use of slice-based measures for assessing the quality of 63 C programs [15]. Their study is based on the following situation: In a system different relations between different components can be detected. These relations make up part of the class of semantic complexity. When taking the information flow within and between these components as quality indicators, then the dual measures of coupling and cohesion are quite descriptive when assessing the quality of a program.

A practical way to calculate the needed measures is to make use of slices. Weiser [26, 27] already introduced five slice based measures for cohesion, and three of them have later on been formally defined by Ott and Thuss [20]: Tightness, Coverage, and Overlap. Coupling, on the other hand, was defined as the number of local information flow entering and leaving a procedure, and Harman demonstrated in [10] that it can be calculated by the use of slices, too.

According to [15, 2:6-2:7], Tightness relates the number of statements common to all slices to the length of the module. It favors concise single thought modules where most of the statements are part of all the slices and thus affect all of the outputs. Coverage on the other hand relates the lengths of the slices to the length of the entire module. It favors large slices but does not require them to overlap and thus to indicate single thought modules. As an example, a module containing two independent slices would result in a value for Coverage of 0.5 but a value for Tightness of 0.0. Overlap measures how many statements are common to all the slices and relates the number to the size of all slices. The result is a measure that is not sensitive to changes in the size of the module, it is

Measure	*Definition*	*Description*
Tightness $\tau(\Psi, \psi)$	$\frac{\mid SP_{int}(\Psi,\psi) \mid}{\mid \psi \mid}$	Tightness τ measures the number of predicates included in every slice.
Coverage $Cov(\Psi, \psi)$	$\frac{1}{n} \sum_{i=1}^{n} \frac{\mid SP_i \mid}{\mid \psi \mid}$	Coverage compares the length of all possible specification slices SP_i ($SP_i \in SP(\Psi, \psi)$) to the length of ψ.
Overlap $O(\Psi, \psi)$	$\frac{1}{n} \sum_{i=1}^{n} \frac{\mid SP_{int}(\Psi,\psi) \mid}{\mid SP_i \mid}$	Overlap measures how many predicates are common to all n possible specification slices SP_i ($SP_i \in SP(\Psi, \psi)$).
Inter – Schema Flow $F(\psi_s, \psi_d)$	$\frac{\mid (SU(\psi_d) \cap \psi_s) \mid}{\mid \psi_s \mid}$	Inter-Schema flow F measures the number of predicates of the slices in ψ_d that are in ψ_s.
Inter – Schema Coupling $C(\psi_s, \psi_d)$	$\frac{F(\psi_s,\psi_d) \mid \psi_s \mid + F(\psi_d,\psi_s) \mid \psi_d \mid}{\mid \psi_s \mid + \mid \psi_d \mid}$	Inter-Schema coupling C computes the normalized ratio of the flow in both directions.
Schema Coupling $\chi(\psi_i)$	$\frac{\sum_{j=1}^{n} C(\psi_i,\psi_j) \mid \psi_j \mid}{\sum_{j=1}^{n} \mid \psi_j \mid}$	Schema Coupling χ is the weighted measure of inter-schema coupling of ψ_i and all n other schemas.

Table 3.1 Coupling and cohesion-related measures for Z specifications as introduced in [5]. SP is the set representing all slices SP_i of a schema ψ in a Z specification Ψ. SP_{int} is the Slice Intersection, representing the set of all predicates that are part of all slices. SU represents the Slice Union of all the slices.

only related to the size of the (single) common thought in the module. Coupling between two modules is calculated by relating the inflow and outflow of a module (with respect to other modules in the program). Inflow and outflow are also calculated by making use of slices. Inter-procedural slices yield those statements of a module that are "outside", and, when examining these sets of statements mutually, their relation can be treated as "information flow". The exact semantics behind the measures (including the definitions and some impediments in calculating them) are explained in more details in the paper of Meyers and Binkley [15].

3.3.3 Specification Slicing

Slicing can be applied to formal specifications, too. The idea was first presented by Oda and Araki [19] and has later been formalized and extended by others [7, 3, 29]. The basic idea is to look for predicates that are part of pre-conditions and for predicates that are part of post-conditions. The general assumption is that (within the same scope) there is a "control" dependency between these predicates. "Data dependency", on the other hand, is defined as dependency between those predicates where data is potentially propagated between them.

Metric Comparison (n=1123)								
Sig.			Pearson		Spearman		Kendall	
	Measure 1	*Measure 2*	*R*	*p*	*R*	*p*	*R*	*p*
Strong	*Tightness*	*Coverage*	0.830	.000	0.907	.000	0.780	.000
Moderate	*Tightness*	*Overlap*	0.809	.000	0.749	.000	0.623	.000
	Size (*LOS*)	*Coupling*	0.589	.000	0.686	.000	0.494	.000
	Size (*LOS*)	*Overlap*	-.557	.000	-.543	.000	-.415	.000
	Size (*LOS*)	*Tightness*	-.541	.000	-.551	.000	-.415	.000
	Coverage	*Overlap*	0.531	.000	0.566	.000	0.437	.000
Weak	*Coupling*	*Overlap*	-.343	.000	-.315	.000	-.239	.000
	Size (*LOS*)	*Coverage*	-.284	.000	-.447	.000	-.326	.000
	Coupling	*Tightness*	-.272	.000	-.262	.000	-.191	.000
	Coupling	*Coverage*	0.006	.829	-.102	.000	-.070	.000

Table 3.2 Pearson, Spearman, and Kendall Tau test values (including significance level p) for the correlation of size and slice-based Z specification measures. Values $\mid R \mid \in [0.8-1.0]$ in the mean are classified as strongly correlated, values $\mid R \mid \in [0.5-0.8)$ are classified as moderately correlated, and values $\mid R \mid \in [0.0-0.5)$ are treated as weakly correlated.

With this concept, slices can be calculated by looking at a (set of) predicates at first and then by including all other dependent predicates.

Recently, sliced-based coupling and cohesion measures have then been mapped to Z by taking the above definitions of Meyers and Binkley as initial points (see Table 3.1 for a summary). Based on the calculation of slice-profiles which are collections of all possible slices for a Z schema, the following measures have been assessed in [5]:

- Tightness, measuring the number of predicates included in every slice.
- Coverage, comparing the length of all possible slices to the length of the specification schema.
- Overlap, measuring how many predicates are common to all n possible specification slices.
- Coupling, expressing the weighted measure of inter-schema coupling (the normalized ratio of the inter-schema flow – so the number of predicates of a slice that lay outside the schema – in both directions).

In [5] it was shown that the measures are very sensitive to semantic changes in Z-schema predicates and that the changes of the values are comparable to their programming counterparts. This includes all types of operations on the specification, especially the addition, deletion or modification of predicates. The next and missing step was to look at a larger collection of sample specifications and assessing their expressiveness. The major objective was to find out which of the measures describe unique properties of a Z specification and which of them are just proxies for e.g. the lines of specification text count (LOS).

In the accompanying study more than $12,800$ lines of specification text in $1,123$ Z schemas have been analyzed and the relevant results of the analysis of the measures are

summarized in Table 3.2. The table shows that three tests have been used: Pearson's linear correlation coefficient, Spearman's rank correlation coefficient, and Kendall's Tau correlation coefficient. The objective was to find out whether each of the measures represents some unique characteristic of the specification or not.

The Pearson's correlation coefficient measures the degree of association between the variables, but it assumes normal distribution of the values. Though this test might not necessarily fail when the data is not normally distributed, the Pearson's test only looks for a linear correlation. It might indicate no correlation even if the data is correlated in a non-linear manner. As knowledge about the distribution of the data is missing, also the Spearman's rank correlation coefficients have been calculated. It is a non-parametric test of correlation and assesses how well a monotonic function describes the association between the variables. As an alternative to the Spearman's test, the Kendall's robust correlation coefficient was used as it ranks the data relatively and is able to identify partial correlations.

The head to head comparison of the measures in Table 3.2 shows that the slice-based measures are not only proxies for counting lines of specification text. In fact, most of the pairs do have a weak or moderate correlation only. So, besides the size of the specification, one can select Coverage, Overlap, and Coupling as descriptors for properties of the specification, but, e.g., skip Tightness as it has the highest values of correlation to most of the other measures.

Meyers and Binkley suggested another measure based on the sizes of the generated slices and called it "deterioration" [15]. This measure has also been mapped to Z in [5] and the basic idea goes back to a simple perception: the less trains of thoughts there are in one schema, the clearer and the sharper is the set of predicates.

When a schema deals with many things in parallel, a lot of (self-contained) predicates are to be covered. This has an influence on the set of slices that are to be generated. When there is only one "crisp" thought specified in the schema, then the slice intersections cover all the predicates. On the other hand, when there are different thoughts specified in it, then the intersection usually gets smaller (as each slice only regards dependent predicates). A progress towards a single thought should therefore appear as a convergence between the size of the schema and the size of its slice-intersection, a divergence could indicate some "deterioration" of the formal specification. This measure seems to be a good candidate for checking our assumption whether specifications do age qualitatively or not, and it defined as follows:

Definition 8 Deterioration. *Let Ψ be a Z specification, ψ_i one schema out of n schemas in Ψ, and $SP_{int}(\psi_i)$ its slice intersection. Then Deterioration ($\delta(\Psi)$) expresses the average module size in respect to the average size of the slice intersections SP_{int}. It is defined as follows:*

$$\delta(\Psi) = \frac{\sum_{i=1}^{n} \mid \psi_i \mid - \mid SP_{int}(\Psi, \psi_i) \mid}{n}$$

Please note that the term "deterioration" as introduced in this paper is neither positive nor negative and one single value of deterioration is of course not very expressive. It just tells about how crisp a schema is. It does not allow for a judgement about the quality of the schema itself. Of course, we could state that all values above a pre-defined value x are to be treated as something unwanted, but it depends on the problem at hand whether we can (and should) allow for such schemas. In all, it merely makes sense to look at the differences in deterioration between two consecutive versions of the specification and thus

to introduce the notion of Relative Deterioration. This measure can be defined in such a way that the relative deterioration is greater than zero when there is a convergence between schema size and slice intersection, and it is negative, when the shears between the sizes get bigger, indicating some probably unintentional deterioration. Relative Deterioration is defined as follows:

Definition 9 Relative Deterioration. *Let Ψ_{n-1} and Ψ_n be two consecutive versions of a Z specification Ψ. Then the relative deterioration ($\rho(\Psi_{n-1}, \Psi_n)$) with $n > 1$ is calculated as the relative difference between the deterioration of Ψ_{n-1} and Ψ_n. It is defined as follows:*

$$\rho(\Psi_n) = 1 - \frac{\delta(\Psi_n)}{\delta(\Psi_{n-1})}$$

3.4 Evaluation

With the set of measures at hand and the reasonable suspicion that specifications do age this paper is now taking a closer look at the development of a real-world specification and the effect of changes onto the measures introduced in Section 3.

3.4.1 Experimental Subject

One of the rare, big publicly available Z specifications is the specification of the Web Service Definition Language ($WSDL$) [22]. The document specifies the Web Services Description Language Version 2.0, which is an XML language for describing Web services. Besides natural language guidance, the specification defines the core language that is used to describe Web services based on an abstract model of what the service offers. Additionally, it specifies the conformance criteria for documents in this language. The reason for focusing on this specification is that, with 2004 onwards, a concurrent versioning system (CVS) has been used. $WSDL$ 1.0 is not available in Z, but from November 2004 till the final release in 2007 139 versions have been checked in. The first revision is an adoption of $WSDL$ 1.0, and then, successively, functionality has been added, modified, or deleted. The final revision contains 814 predicates (distributed over $1,413$ lines of Z text).

This specification is now used so check whether, due to maintenance operations, there are drastic changes in the measures and whether deterioration can be detected or not. The strategy is simply to look at the changes (as documented in the code and in the CVS log files) and to compare them to the obtained values.

3.4.2 The Study

As a first step the CVS log was analyzed. This provided some insights to the types of changes that occurred on the way to the final release. Though there have been several changes influencing the events, the following sections and revisions are noticeable and are considered in more details:

- Up to Revision 1.005 there is a mapping of $WSDL$ version 1.0 to Z. Only minor changes to the specification happen.
- Between revisions $1.005 - 1.007$ there are mostly structural enhancements. Finally, model extensions take place.

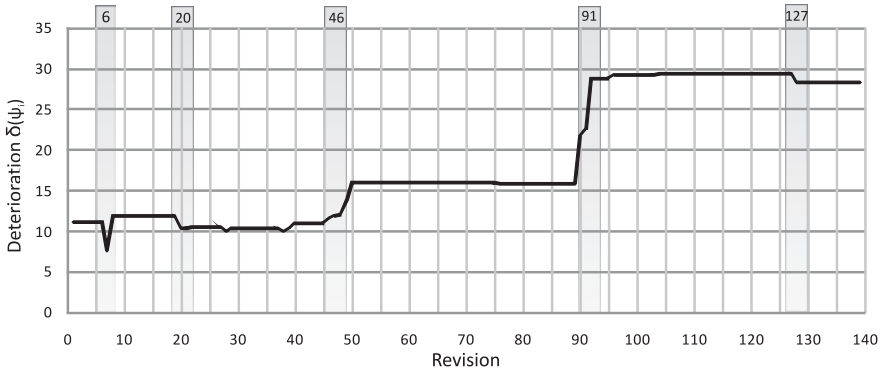


Figure 3.2 Deterioration for the 139 revisions of the WSDL specification. When the value of deterioration increases, then more predicates (not closely related to each other) are introduced to schemas. This is not bad per se, but it is a hint towards a decrease in cohesion values.

- At revisions $1.020ff$ there are several refactoring steps and noticeable extension.
- At revisions $1.045ff$ there are several smaller changes to some of the concepts.
- At revisions $1.090ff$ there are massive extensions to the model and new concepts are introduced.
- Between revisions $1.096 - 1.104$ the concepts in the model are simplified.
- At revisions $1.127ff$ there are change requests and, thereinafter, removing features leads to a structural refactoring.

Up to revision 1.092 the interventions consisted mainly of adding new concepts (in our case Z schemas) to the specification. After revision 1.095 there are solely change requests, leading to a refactoring of the specification. Really massive changes took place at revisions 1.046 and 1.091.

When taking another closer look at the CVS log and the code, a specific strategy for keeping the specification constantly at a high level of quality can be detected. The recurring steps of a change request were:

1. Refactoring of the actual version.
2. Adding, removing, or modifying a concept.
3. Update of the natural language documentation.

The interesting question is now whether our measures introduced in Section 3.3 are able to reflect these changes and whether the measures of deterioration are able to display these changes.

3.4.3 Results

At first let us take a closer look at the measure called deterioration. Fig. 3.2 presents the value for all 139 revisions in the CVS. This figure indicates that the specification

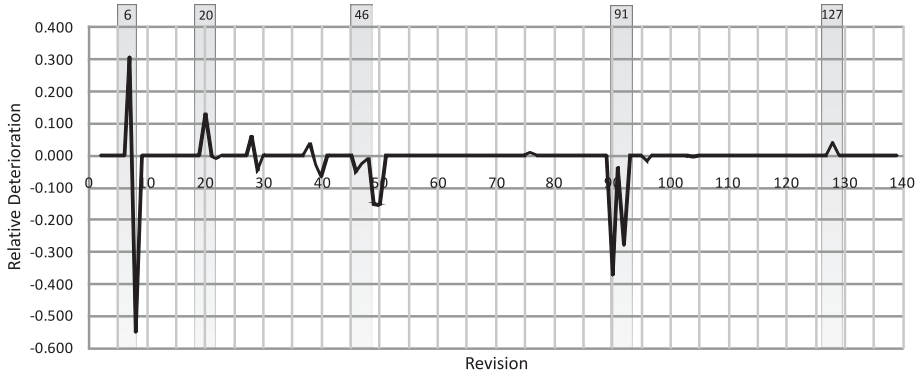


Figure 3.3 The change in deterioration is better visible when looking at the relative deviation over the time. A positive value indicates an increase in cohesion, while a negative value indicates a decrease in the values of cohesion.

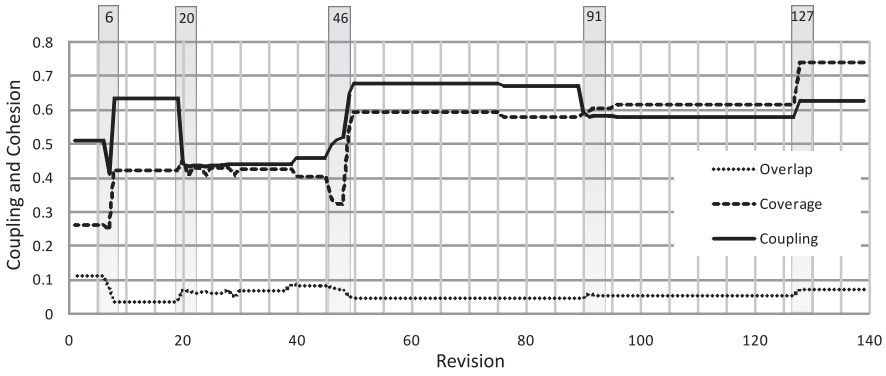


Figure 3.4 The values of cohesion (expressed by the measures of overlap and coverage) and coupling for the 139 revisions of the WSDL specification. In most cases the values are subject to the same changes. However, at revisions 1.006 and 1.044 we observe changes into different directions, too.

remarkably changes at revisions 1.046 and 1.091. In fact, the *CVS* log also documents the changes.

As absolute values (as in Fig. 3.2) do not perfectly describe the influence of a change, the notion of relative deterioration has been introduced in Section 3.3. Fig. 3.3 presents the value of it for all 139 revisions. Positive values indicate that the difference between the schema sizes and their slice intersections is reduced; such a deviation is assumed to be positive in respect to deterioration as the slice intersection is a measure of how strong the predicates are interwoven in a schema. On the other hand, negative values indicate negative effects.

When taking again a look at Fig. 3.3 (especially between revisions 1.020 and 1.046), then the above mentioned strategy of change requests gets noticeably visible. A change is implemented by a structural improvement first (to be seen as a positive amplitude), and then it is followed by the introduction of the new concept, in most cases indicated by a negative amplitude in the diagram.

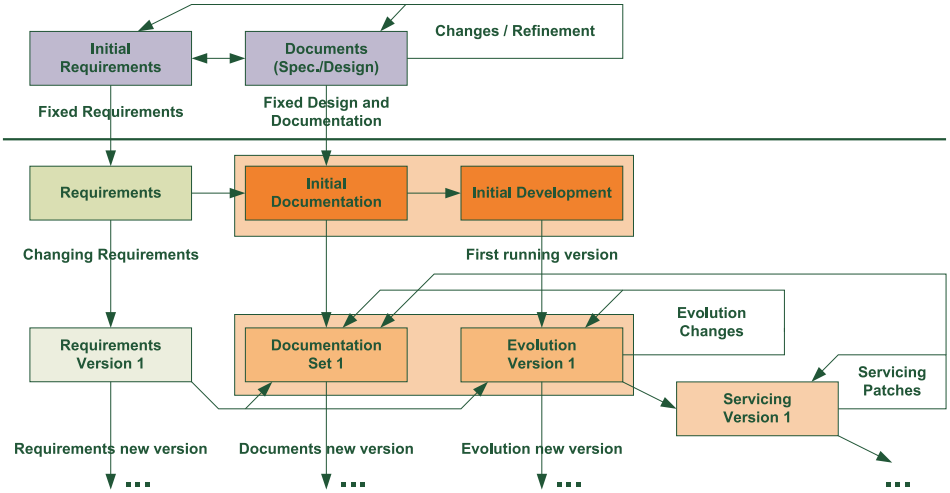


Figure 3.5 A refined and extended look at the versioned stage model. Starting with a first set of initial requirements several versions of documents are created. Requirements are refined, and formal specifications are (among other design documents) also changed and modified. When the design is fixed, development is about to begin. Due to evolutionary changes after this phase, the existing documentation – including specifications and design documents – is (and has to be) changed, too. The term "evolutionary change of a formal specification" is used in a rather general sense. Apart from the classification of system types of Lehman [14], the figure illustrates that essential changes might happen to documents before and after delivery.

Let us now analyze the influence of a change onto the qualitative values of coupling and cohesion. By looking at Fig. 3.4, we see that the value for overlap decreases (on average) a bit. This indicates that, with time, the number of predicates, common to other slices, gets lower. Single Z schemas seem to deal with more independent thoughts. Refactoring these thoughts into separate schemas (which happened e.g. at revisions 1.020 and 1.127) helped a bit to improve the structure of the specification again.

The value of coverage follows more or less the fluctuation of overlap – but not at all revisions to the same extent. On the long run it definitely increases. Coverage tells us about how crisp a schema is, and in our case the developers of the specification did not manage to keep this property stable.

Finally, coupling refers to the flow between different schemas in the specification. Though the value fluctuates, the developers managed to keep coupling quite stable on the long run. Fig. 3.4 also shows that the value fluctuates with the values of cohesion, but not necessarily to the same extent, and not necessarily inversely (as would be assumed to be normal).

Though with the *WSDL* specification there is only one experimental subject, the results seem to substantiate that the measures are suitable for assessing this Z specification. The measures called deterioration and relative deterioration reflect the aging of the system quite well, and the measures for coupling and cohesion (a) do indicate structural changes and (b) also seem to explain some of the semantic changes in the specification.

3.5 An Extended Model of Evolution

As has been shown in Section 3.4.3, specifications keep on changing. Either one is still in the process of finding the most suitable version for our requirements, or one is modifying them due to changes in our projects' software systems. With that, a second look at the software evolution model in Fig. 3.1 is quite helpful – as one comes down to the following statement so far:

There is also evolution before the birth of the running version of the software system.

Fig. 3.5 tries to exemplify this for the initial and evolutionary versions of the software. In this figure the original model has been extended by refining the boxes of the evolutionary versions. Documents and requirements have been added so that formal specifications are made explicit (as they do belong to the set of necessary documents). They are, depending on the changing requirements, also changed. These changes either happen before one has a first running version of the software or afterwards.

The implications of this (refined) picture are manifold and should be considered when using formal specification languages in the software development lifecycle:

- Suitable size- and quality-based measures should be defined. This ensures that changes in the various documents – including formal specifications – can be detected and assessed. The slice-based measures introduced above are just an example of how it could be done for a specific specification language. For other languages the ideas might be reused. It might also be necessary to define new measures. However, the crucial point is that there is a measurement system around.
- Points of measurement should be introduced at every change/refinement loop. This ensures that the effects of changes can be assessed, and that the further direction of the development can be steered. The example of $WSDL$ shows that already during the initial development changes have effects and that it takes effort to keep a specification constantly at a pre-defined level of quality. One can assume that $WSDL$ is not an exception and that the observation also holds for other specification documents. By making use of a measurement system one is at least on the safe side.
- The terms "Fixed Design and Documentation" just designate the conceptual border between the initial development phase and the first running version. Nevertheless, changes to the documents happen before and after this milestone in the project (as evolution is about to begin). The previously introduced measure points should also be defined for the evolutionary phases, and measures should be collected during all the evolutionary changes and servicing activities (influencing the documents and specifications).

Basically, the extended model of software evolution makes one property of specification (and other documents) explicit: they are no exception to aging. With this, it is obvious that measures, at the right point and extent, help in answering the question of what happened and, eventually, of what can be done for preventing unwanted deterioration effects.

3.6 Conclusion

We started with the observation that formal specifications – documents important in the very early phases of software development and later on during maintenance – might be

changed more often than commonly expected and that the changes are not necessarily positive. We wanted to verify this observation, and so we mapped existing semantic-based measures to Z specification and used them to analyze a large real-world specification. The lessons learned so far are manifold.

1. Firstly, there was no suitable measurement system around. In order to understand and to assess changes, new measures had to be developed. These measures, carefully mapped to Z specifications, have been evaluated by making use of a large set of sample specifications. They are maybe not representative for all specifications around, but the statistical tests helped us in gaining at least basic confidence in the results for Z.

2. Secondly, changes onto formal specifications definitely might influence the values of the measures and there is the chance that their effects are underestimated. There is no model that enunciates this situation, which is also the reason why we borrowed from the model of evolution and refined it to cover the phases before, within, and after initial development. A closer look at the evolution of the $WSDL$ specification seems to confirm the observation mentioned at the beginning of the paper: formal specifications are not just written once. They are modified, are extended, and they age.

3. Finally, the measures of coupling and cohesion (and with them deterioration and relative deterioration) seem to be a good estimate for a qualitative assessment of a specification. They are easy to calculate and seem to point out a possible loss in quality.

With that, we are able to answer our two questions that have been raised at the end of Section 3.1: specifications evolve and this evolution can be observed by simple semantics-based measures. The refined and extended model of evolution as presented in Section 3.5 is a good image of what happens when developing our systems.

The results of this contribution are interesting insofar as it turned out that, for the full benefits of a formal software development process, it makes sense to permanently take care of the quality of the underlying formal specification(s). Even when declarative specification languages are used, this can easily be done by defining suitable measures and by using them to constantly monitor the quality of the whole system. The goals for future work now include (a) taking a closer look at other formal specifications in order to verify and consolidate the findings, (b) investigating the correlation of the specification measures to code-based measures in order to come up with different prediction models, and (c) incorporating the refined model of software evolution into a formal software development process model that also considers cultural differences between the different stakeholders in a project.

Overall, the results are encouraging. Formal specifications are not necessarily restricted to the early phases of a software development process. When treated carefully (and kept up to date) they may help us in producing software systems that can be trusted, even when changed.

Acknowledgment

I am grateful to the reviewers of the FM 2011 conference and to my colleagues at AAU Klagenfurt, especially to Prof. Mittermeir, who helped me with fruitful discussions and reflections on this topic.

REFERENCES

[1] Keith Bennet and Vaclav Rajlich. Software Maintenance and Evolution: a Roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–89. ACM New York, NY, USA, 2000.

[2] Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal Versus Agile: Survival of the Fittest. *IEEE Computer*, 42(9):37–54, September 2009.

[3] Andreas Bollin. *Specification Comprehension – Reducing the Complexity of Specifications*. PhD thesis, AAU Klagenfurt, April 2004.

[4] Andreas Bollin. Concept Location in Formal Specifications. *Journal of Software Maintenance and Evolution – Research and Practice*, 20(2):77–105, 2008.

[5] Andreas Bollin. Slice-based Formal Specifiation Measures – Mapping Coupling and Cohesion Measures to Formal Z. In Cèsar Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium*, NASA/CP-2010-216215, pages 24–34. NASA, Langley Research Center, April 2010.

[6] David Carrington, David Duke, Ian Hayes, and Jim Welsh. Deriving modular designs from formal specifications. In *ACM SIGSOFT Software Engineering Notes*, volume 18, pages 89–98. ACM, December 1993.

[7] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[8] Edmund M. Clarke and Jeannette M.Wing. Formal Methods: State of the Art and Future Directions. Technical report, Carnegie Mellon University, CMU-CS-96-178, 1996.

[9] B. P. Collins, J. E. Nicholls, and I. H. Sorensen. Introducing formal methods: the cisc experience with z. In *Mathematical Structures for Software Engineering*, pages 153–164. Clarendon Press Oxford, 1991.

[10] Mark Harman, Margaret Okulawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of coupling. In *Proceedings of the IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution. Boston, Massachusetts*, pages 28–32, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[11] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):1–76, 2009.

[12] Mike Hinchey, Michael Jackson, Patrick Cousot, Byron Cook, Jonathan P. Bowen, and Tiziana Margaria. Software engineering and formal methods. *Communications of the ACM*, 51(9):54–59, September 2008.

[13] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. The MIT Press, Cambridge, Massachusetts, 1996.

[14] Meir M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1(1):213–221, 1979.

[15] Timothy M. Meyers and David Binkley. An Empirical Study of Slice-Based Cohesion and Coupling Metrics. *ACM Transactions on Software Engineering and Methodology*, 17(1):2:1–2:27, December 2007.

[16] Roland T. Mittermeir and Andreas Bollin. Demand-Driven Specification Partitioning. *Lecture Notes in Computer Science*, 2789(2003):241–253, 2003.

[17] Roland T. Mittermeir, Andreas Bollin, Heinz Pozewaunig, and Dominik Rauner-Reithmayer. Goal-Driven Combination of Software Comprehension Approaches for Component Based Development. In *Proceedings of the ACM Symposium on Software Reusability - SSR01*, volume 26 of *Software Engineering Notes*, pages 95–102. ACM Press, 2001.

[18] Juan C. Nogueira, Luqi, Valdis Berzins, and Nader Nada. A formal risk assessment model for software evolution. In *Proceedings of the 2nd International Workshop on Economics-Driven Software Engineering Research (EDSER-2)*, 2000.

[19] Tomohiro Oda and Keijiri Araki. Specification slicing in a formal methods software development. In 17^{th} *Annual International Computer Software and Applications Conference*, IEEE Computer Socienty Press, pages 313–319, November 1993.

[20] Linda M. Ott and Jeffrey J. Thus. The Relationship between Slices and Module Cohesion. In *11th International Conference on Software Engineering*, pages 198–204, Los Alamitos, CA, USA, 1989. IEEE Computer Society.

[21] Helfried Pirker, Roland Mittermeir, and Dominik Rauner-Reithmayer. Service Channels - Purpose and Tradeoffs. In *COMPSAC '98 Proceedings of the 22nd International Computer Software and Applications Conference*, pages 204–211, 1998.

[22] Sun Microsystems Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. http://www.w3.org/TR/wsdl20, 2007.

[23] Philip E. Ross. The Exterminators. *IEEE Spectrum*, 42(9):36–41, September 2005.

[24] W.B. Samson, D.G. Nevill, and P.I. Dugard. Predictive software metrics based on a formal specification. In *Information and Software Technology*, volume 29 of *5*, pages 242–248, June 1987.

[25] Rick Vinter, Martin Loomes, and Diana Kornbrot. Applying software metrics to formal specifications: A cognitive approach. In *5th International Symposium on Software Metrics*, pages 216–223, Bethesda, Maryland, 1998. IEEE Computer Society.

[26] Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.

[27] Mark Weiser. Program slicing. In *Proceedings of the* 5^{th} *International Conference on Software Engineering*, pages 439–449, Piscataway, NJ, USA, 1982. IEEE Press.

[28] Jim Woodcock and Jim Davis. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, July 1996.

[29] Fangjun Wu and Tong Yi. Slicing Z Specifications. *ACM SIGPLAN Notices*, 39(8):39–48, 2004.

CHAPTER 4

METRICS FOR QUANTIFYING EVOLUTIONARY CHANGES IN Z SPECIFICATIONS

A. Bollin

To appear in Software Maintenance and Evolution: Research and Practice. Wiley and Sons Ltd., 28 pages, Accepted June 2012.

4.1 Abstract

This article proposes metrics for quantifying changes throughout the evolution of formal software specifications in long living systems. Formal specifications play an important role in the software development life-cycle by supporting refinement and proof and by providing a basis for comprehension and maintenance activities. However, specifications also undergo evolutionary changes, and these changes are hard to assess due to a lack of suitable measures.

This paper proposes and analyzes a set of measures for estimating aspects of a specification's complexity and quality. The measures are based on existing measures for source code, but they have been redefined in the scope of formal Z specifications. Geared towards Z, they are then evaluated concerning their expressiveness by a case study that comprises more than 65,000 lines of specification text. Finally, the usability of the measures is demonstrated on the Z specification of the Web Service Definition Language during its evolution over a period of about three years.

4.2 Introduction

Formal specifications can be very useful artifacts at various stages of software development. They enable verification and validation, but they also form the basis for ongoing refinement steps [30]. Though there are several myths around [22, 9], there are also a lot of reasons for using them in order to improve the quality of the systems to be generated [52, 43, 27, 26].

Another, sometimes underestimated, benefit arises when specifications are kept up-to-date during the evolutionary stages of the development. This facilitates ongoing maintenance activities. Since formal specifications are close to the requirements, these documents are usually descriptions at a high level of abstraction and thus accelerate the underlying comprehension process.

The use of formal specifications looks like a silver-bullet, but the benefits are also put into question. It is interesting to ponder the reasons for this situation. The old management adage "You can't manage what you don't measure" might be a hint towards an answer. Fenton and Kaposi [19, p. 7] already mention it when they write that *[...] in the absence of a suitable measurement system, there is no chance of validating the claims of the formal methods community [...].* Hall et. al also state that formal methods do have a lot of benefits, but they also admit that it really is *difficult to be more precise* [23, p. 22].

Since the 1990s the situation has definitely improved. But, compared to the number of articles regarding software measures, the amount of studies dealing with specification measures is still small. As mentioned in Section 4.7, Related Works, developers basically do count lines of specification text or the number of predicates. But other measures closer to the aspects of the semantics of a specification are not in use. One of the impediments when trying to reuse existing measures is the structural difference between formal specification languages and programming languages. Due to the declarative nature of specifications, concepts like control- or data-flow are not predominant. But, these concepts are very often used to form the basis for wide-spread software measures, and so it becomes clear why it is not easy to apply "traditional" software measures to formal specifications.

This contribution tries to demonstrate that this situation can be changed. With the reconstruction of control and data dependencies [37] "classical" measures can be mapped to specifications, too. Bollin [6] demonstrated that slice-based coupling and cohesion measures can be reasonably mapped to formal Z specifications. Using them, the existing set of measures can be extended by semantics-based measures, too. What is left (and will be shown in this paper) is to check whether these measures are unique views onto the specifications, to ensure that they are not only proxies for size/quantity-based measures, and that they can be used to talk about possible deterioration effects.

Motivated by this situation and by a study by Meyers and Binkley [35] where they are looking at coupling and cohesion-based software measures, small, medium, and large-size Z specifications have been collected and used for a thorough evaluation of the different measures. The evaluation is intended to be comparable to the study by Meyers and Binkley, and thus this paper follows the structure of their work for the most part.

In total, more than 65,000 lines of specification text have been taken as a basis for this study. The main steps of this study are:

- The study compares size-/structure- and semantics-based measures head to head. The comparison shows which of these measures are strongly related and which of them are

providing unique views of the specification. Section 4.5.3 shows that there really are measures that are not just proxies for counting lines (or predicates) in a specification text.

- The paper analyzes the effect of evolutionary changes of the specification with respect to the measures. One longitudinal study, based on the specification of the Web Service Definition Language, is presented. Section 4.5.4 shows that some of the measures are quite sensitive to changes in the specification.
- Finally, in Section 4.5.5 the study presents baseline values for the measures, depending on the sizes of the specifications under study.

This paper is structured as follows. Section 4.3 provides the necessary terminology for the study. Ensuing, a set of existing measures is presented and discussed in Sec. 4.4. Then, based on a large set of sample specifications, in Section 4.5 the measures are evaluated and assessed in respect to their expressiveness. Possible threads regarding validity are discussed in Section 4.6. Related work is discussed in Section 4.7, before Section 4.8 summarizes the most important findings.

4.3 Measurement Basis

The objective of this section is to introduce the elements and structures of a formal specification being measured. Most of the measures are calculated on the basis of specification dependencies and slices, so this section addresses these concepts in some details.

4.3.1 Specification Slicing

Static program slicing was introduced by Weiser [50, 51] in order to compute those set of program statements that affect program values at some point of interest (called slicing criterion). Since the original definition there has been a substantial development of the concept (an introduction can be found in the overview paper by Tip [49]), but basically, static slices are computed by transitively related program statements according to control and data dependencies.

As control and data dependencies are not predominant concepts in formal specifications, slicing was restricted to programming languages for a while. In 1993, Oda and Araki [38] then transferred the concept of slicing to specifications by defining a static forward slice for a formal specification based on data dependencies. One year later, Chang and Richardson [12] extended this idea by also considering control dependencies. Their approach rests on a simple idea: predicates of a specification that describe an after-state are "control"-dependent on predicates that do not refer to an after-state. "Data dependency", on the other hand, is defined as dependency between predicates potentially connected by data flow. By removing all the predicates that are not dependent upon a point of interest (which is one predicate or a set of predicates in the specification), static specification slices are carved out of the specification.

The approach of Chang and Richardson has been refined and extended by Mittermeir and Bollin [37, 3] in 2003. There, the different dependencies have been defined formally, and also the procedure for carving out the elements and dependencies has been specified in an algorithmic and semantics preserving manner. The approach finally yielded a Java

Line	Birthday Book Specification (BB)	$SP^{fb}(BB, Add)$		$SP^{fb}_{\cap}$	$SP^{fb}_{\cup}$
1	$[NAME, DATE, GIFT]$				
2	BB				
3	$known : \mathbb{P}\, NAME$				
4	$birthday : NAME \pfun DATE$				
5	$gift : NAME \pfun \mathbb{P}\, GIFT$				
6	$known = \text{dom}\ birthday$	|	|	|	|
7	$\text{dom}\ gift \subseteq known$	|	|	|	|
8					
9					
10	Add				
11	ΔBB				
12	$name? : NAME$				
13	$date? : DATE$				
14	$name? \notin known$	|	|	|	|
15	$birthday' = birthday \cup$	×			|
16	$\{name? \mapsto date?\}$	×			|
17	$gift' = gift$		×		|
18					
19					

Table 4.1 Example of a simple Z specification (a slightly modified and incomplete version of the Birthday-Book specification [47]) including its related slice profile $SP^{fb}(BB, Add)$, the slice intersection $SP^{fb}_{\cap}(BB, Add)$ and slice union $SP^{fb}_{\cup}(BB, Add)$. Predicates that are part of the slice profile, the intersection or union are marked by a bar. The slicing criteria are marked by a cross. See Section 4.3.2 for more details.

prototype called ViZ that is able to generate various types of partial formal specifications [5], including those of specification slices.

The major difference to the approach of Chang and Richardson is that the ViZ environment, in a first step, maps the specification to a graph and then, in a second step, annotates it with different types of dependencies. Alternative approaches, like that of Wu and Yi [54], also make use of a graph-based approach.

Table 4.1 and Figure 4.1 visualize the basic idea behind the mapping process and the calculation of slices. The left part of Table 4.1 presents a simple (and for demonstration purposes shortened and thus incomplete) formal Z specification – a slightly modified and shortened version of the Birthday Book specification [47] that stores names, related birthdays and gifts in a "database" (represented as sets and functions). At line 1 it introduces three basic types namely $NAME$, $DATE$, and $GIFT$. Then, two boxed schemas are provided. The first one (between lines 2 and 9) is called BB (for Birthday Book) and it represents the state of the system. The second one (between lines 10 and 19) is called Add, and it represents an operation schema for adding new entries to the set of names. Both boxed schemas are divided into two parts (noticeable by the horizontal lines at lines 6 and 14). It separates the declaration part from the predicate part. The declaration part of the state schema introduces three basic declarations written on separate lines: a set of known names at line 3, a partial function mapping names to birth dates at line 4, and a partial function mapping names to a set of birthday-presents at line 5. The predicate part

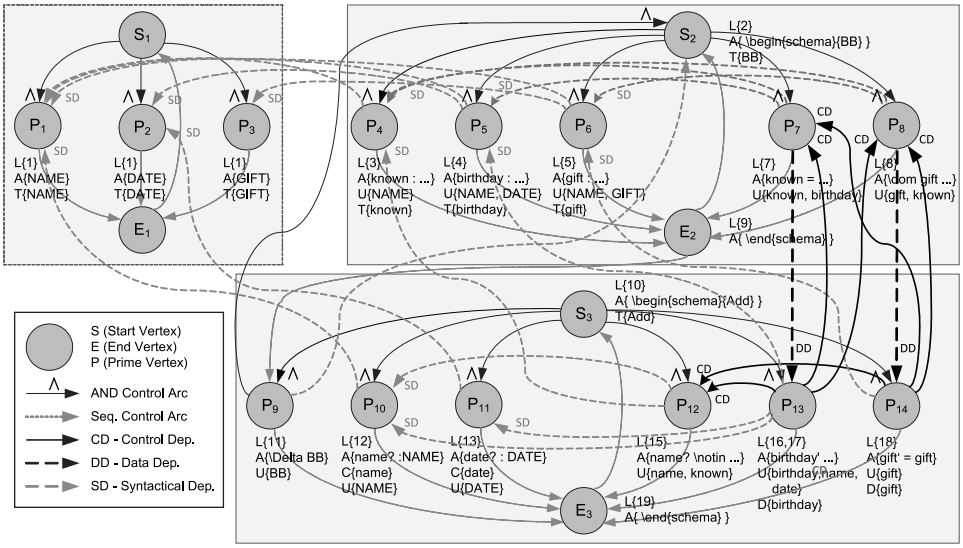


Figure 4.1 *ASRN* graph [37] for the Birthday Book specification presented in Table 4.1. Predicates and declarations are represented by Prime vertices (P). They are enclosed between start (S) and end (E) vertices and are additionally annotated by position/line numbers, the related Z specification text, and the sets describing the use of the identifiers (D..defined, U..used, T..type defined, C..in/out channel). Logical combinations are expressed by logical control arcs - in our example AND control arcs.

of the state schema BB contains two predicates (state invariants), ensuring that, for the names stored in the database, the birthday dates are defined, and ensuring that presents are defined for existing entries only. At lines 12 and 13 two identifiers, namely $name?$ and $date?$, are defined. The question mark indicates that they are input identifiers in the operation schema. The declaration part of the Add schema introduces another feature of the Z specification language: it is possible to include another schema in the declaration part. At line 11 the state schema BB is included which means that the declarations (and predicates) of the included schema are now considered to be part of the actual schema. By following a naming convention, "$\Delta\ BB$" means that the state of BB might change due to the operation schema[1]. In the predicate section there are three predicates. The first one at line 15 ensures that the name taken as input is not part of the database already. The second and third (at lines 16 to 18) describe the after-state of the two functions $birthday$ and $gift$, indicated by the decorated identifiers. An in-depth introduction to the syntax and semantics of Z can be found in the textbook by Diller [17].

The *ViZ* environment makes use of the *CZT* parser [33] for identifying the basic elements a specification is built upon: specification primes, comparable to program statements. A specification prime represents the basic entity of a specification - it consists of specification literals and forms logical, syntactic, or sematic units. For Z specifications a prime is either (a) a given set or (b) a free-type definition, (c) a declaration or a (d) (sub-)predicate (separated by a logical AND or OR from other predicates). A more detailed introduction into the topic of specification primes and their identification can be found in

[1]For indicating that a state does not change, usually the notation "$\Xi\ Schemaname$" is used.

[3, p.43]. In Z specifications they are identified by traversing the AST generated by the CZT parser and by looking for those sub-trees that represent names, predicates, and expressions. In the example in Table 4.1 these primes are the three type definitions (line 1), the six schema declarations (lines 3 to 5 and 11 to 13), and the five predicates (lines 7 and 8, line 15, lines 16-17, and line 18). These elements are mapped to a graph as presented in Figure 4.1. The graph, called Augmented Specification Relationship Net ($ASRN$), consists of vertices called "prime vertices" representing the basic elements of the specification (in our case all definitions, declarations, and predicates of the BB specification), and arcs representing the various types of dependencies between them[2]. The $ASRN$ in Figure 4.1 has already been annotated with its dependency arcs. For example, there is a control dependency between vertex P_{13} and P_{12} or a data dependency between P_7 and P_{13}

Slices are typically defined for a specific point of interest. This "point" is, according to similar definitions in the slicing community, called the *Abstraction Criterion*. For Z, it consists of a set of primes to be regarded and the types of dependencies to be taken into account. For example, when starting with P_{13} and including all primes that are reachable via control and data dependency arcs (in a forward and a backward manner), then the resulting set of primes is $\{ P_7, P_8, P_{12}, P_{13} \}$. When also including primes that the primes in the slice are syntactically dependent on (by following the syntactical dependency arcs in the $ASRN$) then the resulting set is treated as a syntactically correct specification slice for P_{13}.

4.3.2 Slice-Profiles

A slice-profile is simply the collection of all possible slices for a given piece of (SW-) code. Various authors demonstrate that this collection of slices can be used to calculate coupling and cohesion measures of procedural programs [40, 24, 35]. Mapping this idea to formal specifications first means to specify the slices and their intersections to be generated.

However, before providing the definitions for slices and slice-profiles, a couple of decisions have to be made. Longworth, and also Meyers and Binkley [32, 35], mention a couple of decisions to be taken when calculating slice-profiles for ordinary programs. These decisions address the the total number of slices to be generated, the type of slices to be calculated, and the question of correct selection of the abstraction criteria. Their considerations also affect the way of producing slices for formal Z specifications. The reason is that also in the case of specifications the dependencies (and thus the arcs in the $ASRN$) do have a direction. Thus the generation of slices can also be done in different ways: forward and/or backward, transitively or not.

A closer look into these problems shows that for cohesion values it makes sense to calculate *transitive forward and transitive backward* slices. This holds because there is no ordering of the predicates in a schema and an artificially introduced "direction" (comparable to the notion of $VRES$ slices defined by Ott and Thuss [40]) would be semantically wrong. Additionally, a single slice for only the "last" predicate in a schema is not appropriate. There is no "order of execution" in a schema that tells us which of the predicates to be evaluated first and which are to be evaluated next. So, all predicate primes have to be regarded, and slices have to be generated for all primes that either do change or might

[2]Please note that the transformation between a specification and the $ASRN$ is one-to-one [3]. When talking about a specification it can be either the textual or the $ASRN$ representation.

change the state. Concerning the $ASRN$, this means that all primes (vertices) representing predicates have to be taken as a slicing criterion.

Concerning the value of coupling, it turns out that the same slice type as introduced by Harman et. al. [24] can be used. In order to include all primes that might eventually influence a given prime in another schema, the slice is calculated in a *transitive* manner, and as the calculation of coupling is based on the notion of information flow (which implies a given direction), the slice to be taken is the *transitive backward slice*. For the calculation of the values, slices and slice profiles are needed. The definitions are given as follows:

Definition 10 Transitive Forward Slice. *Let Ψ be a formal Z specification, ψ one schema out of Ψ, and V a set of primes in ψ called $Abstraction\ Criterion$. $SSlice_f(\Psi, \psi, V)$ is called* transitive forward slice *of ψ for primes V. It contains the primes in V and all primes p of the specification ($p \in \Psi$) that are (transitively) reachable from the primes in V by following control and data dependencies.*

Definition 11 Transitive Backward Slice. *Let Ψ be a formal Z specification, ψ one schema out of Ψ, and V a set of primes in ψ called $Abstraction\ Criterion$. $SSlice_b(\Psi, \psi, V)$ is called* transitive backward slice *of ψ for primes V. It contains the primes in V and all primes p of the specification ($p \in \Psi$) that are (transitively) connected to primes in V by data or control dependency arcs.*

Definition 12 Transitive Forward/Backward Slice. *Let Ψ be a formal Z specification, ψ one schema out of Ψ, and V a set of primes in ψ called $Abstraction\ Criterion$. $SSlice_{fb}(\Psi, \psi, V)$ is called* transitive forward backward slice *of ψ for primes V. It consists of all primes either in $SSlice_f(\Psi, \psi, V)$ or in $SSlice_b(\Psi, \psi, V)$.*

Definition 13 Slice Profiles. *Let Ψ be a formal Z specification and ψ one schema out of Ψ. Additionally, let V be the set of primes v in the $ASRN$ representing predicate primes in ψ. The collection of all transitive forward backward slices ($SSlice_{fb}(\Psi, \psi, \{v\})$ with $v \in V$) for a schema is called $forward\ backward\ Slice\ Profile$ $SP^{fb}(\Psi, \psi)$. The collection of all transitive backward slices ($SSlice_b(\Psi, \psi, \{v\})$ with $v \in V$) is called $backward\ Slice\ Profile$ $SP^b(\Psi, \psi)$.*

Both types of slice profiles just differ in the type of slices used for their calculation. In order to improve the legibility of the text, both types of profiles ($SP^{fb}(\Psi, \psi)$ and $SP^b(\Psi, \psi)$) will from now on just be called $SliceProfile$. Only when necessary, the full name will be given.

Definition 14 Size, Slice Intersection, Smallest Slice, Largest Slice, Slice Union. *Let Ψ be a formal Z specification and ψ one schema out of Ψ. Let V be the set of primes v in the $ASRN$ representing predicate primes in ψ. Additionally, let $SP^x(\Psi, \psi)$ be the $SliceProfile$ of schema ψ (where "x" stands either for "fb" or "b"). Then it holds:*
(a) $SP^x_i(\Psi, \psi)$ denotes one specific slice out of the collection of slices in $SP^x(\Psi, \psi)$ ($1 \leq i \leq \mid V \mid$).
(b) The number of slices in $SP^x(\Psi, \psi)$ is called the size if the slice profile $\mid SP^x(\Psi, \psi) \mid$.
(c) $SP^x_{min}(\Psi, \psi)$ denotes the smallest slice in the slice profile $SP^x(\Psi, \psi)$.
(d) $SP^x_{max}(\Psi, \psi)$ denotes the largest slice in the slice profile $SP^x(\Psi, \psi)$.
(e) The intersection of all slices in the slice profile is called $Slice\ Intersection$ ($SP^x_\cap(\Psi, \psi)$).
(f) The union of all slices in the slice profile is called $Slice\ Union$ ($SP^x_\cup(\Psi, \psi)$).

Table 4.1 (right part) presents a simple example of a slice profile for the Add operation schema of the Birthday Book specification. At first, forward backward slices are generated for the two predicates at lines 16/17 and 18. The resulting slices are then part of the Slice Profile $SP^{fb}(BB, ADD)$. The intersection of the sets of primes yields the Slice Intersection $SP^{fb}_{\cap}(BB, Add)$, and the union of all the primes yields the Slice Union $SP^{fb}_{\cup}(BB, Add)$.

4.4 Specification Measures

As discussed in the related work section, most of the studies focus on size/quantity-based measures. They are easy to calculate and give a good glimpse of parts of the complexity of the underlying document. However, with the $ASRN$ and its explicitly available dependencies further possibilities exits. It enables structure as well as semantics-based considerations. The definitions in the following section are given for formal Z specifications. It would go beyond the scope of this paper, but for other declarative, state-based formal specification languages a mapping should be possible, too. The measures are defined upon the $ASRN$, and all that has to be done is to map the basic elements (predicates) to the graph-based structure, and then to annotate the graph with occurring dependencies.

4.4.1 Classes of Specification Measures

Due to simplicity of assessing values, two classes of measurement are commonly used since the 1970s; those based on the physical size of the source text, and those based on the analysis of data and control flow. In addition to these classes, qualitative measures have been developed as a third class in order to describe the semantic compactness of components or functions of a system. The first two classes describe quantitative properties, whereas the third class focuses on qualitative aspects of the underlying artifact.

Class (1). The first set of measures deals with size or structure-based complexity considerations. There, quantity (or size)-based measures only consider the literals without a deeper interpretation of the text itself. A wide-spread metrics of this type is the number of lines of code (LOC). But, as lines of specification text are not necessarily very informative, it is more convenient to count the number of predicates or expressions.

Class (2). Structural metrics, on the other hand, usually refer to the relations between the individual sections. Two well-known (and also widely discussed) metrics are that of cyclomatic complexity $v(G)$ going back to McCabe [34], and the $Definition - Use$ count metric of Tai [48]. Structural metrics are often based on control and/or data flow relationships. As these dependencies are made explicit in the $ASRN$, they can also be calculated for Z specifications.

Size and structure-based measures will be considered briefly in Section 4.4.2 and are discussed in more detail by Bollin [4].

Class (3). Quality considerations are said to be crucial during a software project, but the term “quality” strongly depends on the perspective on is looking upon it. Within the scope of this contribution we focus on quality measures that describe the semantic compactness of components or functions of the underlying system. Quite often they are expressed by the dual properties of coupling and cohesion, and this study also assesses these types of measures. A summary of quality-based specification measures is given in Section 4.4.3. A more detailed discussion is presented by Bollin [6].

Measure	*Definition*
Conceptual Complexity: $CC(\Psi)$	$\#\{v : \Psi \mid v.type = Prime\}$
Logical Complexity: $v'(\Psi) = (l, u)$	$l = 1 + \#\{v : \Psi \mid \exists\, a : Arc \bullet a.type = CD \land a.dest = v\}$ $u = 1 + \#\{a : Arc \mid \Psi\ contains\ a \land a.type = CD\}$
Definition-Use Count: $DU(\Psi)$	$\#\{a : Arc \mid \Psi\ contains\ a \land a.type = DD\}$

Table 4.2 Size and structure based measures for Z specifications Ψ.

4.4.2 Complexity Metrics

Complexity can be seen as the difficulty of describing or understanding parts/pieces of artifacts, in our case formal Z specifications. It would be tempting to define a single value for complexity. However, complexity cannot be reduced to just one dimension. When being interested in defining measures that influence the human "costs" for the creation and understanding of a specification, then *conceptual and logical complexity* are to be considered. The following measures for these types of complexity have been suggested by Bollin [4] and their definitions are summarized in Table 4.2:

- *Conceptual Complexity* $CC(\Psi)$ of a specification Ψ. This measure equals the total number of prime vertices in the $ASRN$.).
- *Logical Complexity* $v'(\Psi)$ of a specification Ψ. Based on the number of control dependencies, lower and upper bounds for (logical) decisions in a specification are counted. The lower bound value l is 1 plus the number of primes that are terminal vertices of control dependence (CD) arcs in the $ASRN$. The upper bound value u equals 1 plus the total number of control dependencies (CD) in the $ASRN$. The lower bound of the measure counts the different decision statements in a specification whereas the upper bound counts all logical dependencies.
- The *Definition-Use Count* metric $DU(\Psi)$ of a specification Ψ. By counting the number of data dependencies the maximum number of data relationships is identified.

4.4.3 Qualitative Measures

Weiser proposed several slicing metrics [51] but did not relate them to cohesion. He assumed a relation to cohesion, but did not examine this in detail. Longworth [32] extended this idea and Ott and Thuss1[40] formally defined them by making use of slice-profiles. The following measures of Ott and Thuss are redefined for formal Z specification schemas by Bollin [6] and also make use of slice profiles:

Measure	*Definition*
Tightness: $\tau(\Psi, \psi)$	$\frac{\mid SP^{fb}_{\cap}(\Psi,\psi) \cap \psi \mid}{\mid \psi \mid}$
Coverage: $Cov(\Psi, \psi)$	$\frac{1}{n} \sum_{i=1}^{n} \frac{\mid SP^{fb}_{i}(\Psi,\psi) \cap \psi \mid}{\mid \psi \mid}$ (with $n = \mid SP^{fb}(\Psi, \psi) \mid$)
Minimum Coverage: $Cov_{min}(\Psi, \psi)$	$\frac{1}{\mid \psi \mid} \mid SP^{fb}_{min}(\Psi, \psi) \cap \psi \mid$
Maximum Coverage: $Cov_{max}(\Psi, \psi)$	$\frac{1}{\mid \psi \mid} \mid SP^{fb}_{max}(\Psi, \psi) \cap \psi \mid$
Overlap: $O(\Psi, \psi)$	$\frac{1}{n} \sum_{i=1}^{n} \frac{\mid SP^{fb}_{\cap}(\Psi,\psi) \cap \psi \mid}{\mid SP^{fb}_{i}(\Psi,\psi) \cap \psi \mid}$ (with $n = \mid SP^{fb}(\Psi, \psi) \mid$)

Table 4.3 Cohesion Measures for a schema ψ in a Z specification Ψ as introduced by Bollin [6].

- *Tightness* $\tau(\Psi)$ is defined as the ratio of the number of primes in the intersection of all slices over the total number of prime elements in the schema.
- *Coverage* ($Cov(\psi)$) relates the number of primes in all n specification slices (in $SP^{fb}(\Psi, \psi)$) to the number of primes in ψ.
- *Minimum Coverage* ($Cov_{min}(\psi)$) is calculated by taking the number of primes in the smallest slice and relating it to the number of primes in the schema.
- *Maximum Coverage* ($Cov_{max}(\psi)$) is calculated by taking the number of primes in the largest slice and relating it to the number of primes in the schema.
- *Overlap* $O(\psi)$ considers the size of the slice intersection and determines how many primes are common to all n slices in $SP^{fb}(\Psi, \psi)$).

Table 4.3 provides the definitions of these measures. As the sizes of the slices are always related and restricted to the size of the schema, the resulting value is between $[0\,..\,1]$. Please note that, in order to reduce calculation complexity and to avoid the "noise" of declarations as mentioned by Ott and Thuss [40], the measures are defined such that only primes representing predicates in the $ASRN$ are considered. The cardinality operator $\mid\,\mid$ for a schema (or set of primes) is therefore defined as follows:

Definition 15 Cardinality. *Let Ψ be a formal Z specification and S be any set of primes out of Ψ. Then the cardinality of S is defined as follows:*

$$\mid S \mid \; = \#\{p : S \mid p.type = P \wedge p \text{ represents a predicate prime in } \Psi\}$$

The calculation of the value of coupling follows the definitions of Harman et. al. [24]. It has also been mapped to formal Z specifications by Bollin [6]. The approach is simple:

Measure	*Definition*
Inter-Schema Flow: $F(\Psi, \psi_s, \psi_d)$	$\frac{\lvert(SP^b_{\cup}(\Psi,\psi_d)\cap\psi_s)\rvert}{\lvert\psi_s\rvert}$
Inter-Schema Coupling: $C(\Psi, \psi_s, \psi_d)$	$\frac{F(\Psi,\psi_s,\psi_d)\times\lvert\psi_s\rvert + F(\Psi,\psi_d,\psi_s)\times\lvert\psi_d\rvert}{\lvert\psi_s\rvert+\lvert\psi_d\rvert}$
Schema Coupling: $\chi(\Psi, \psi_i)$	$\frac{\sum_{j=1}^{n} C(\Psi,\psi_i,\psi_j)\ \lvert\psi_j\rvert}{\sum_{j=1}^{n}\times\lvert\psi_j\rvert}$

Table 4.4 *Coupling* and flow-related measures for a schema ψ_i in a Z specification Ψ (consisting of n specification schemas).

first, the inter-schema flow between two schemas is specified and then taken as the basis for the calculation of the flow between all the schemas in the specification. This leads to the following set of measures (summarized in Table 4.4) defined for formal Z specifications:

- *Inter-Schema Flow* $F(\Psi, \psi_s, \psi_d)$ which measures the number of primes of the slices in ψ_d that are in ψ_s,
- *Inter-Schema Coupling* $C(\Psi, \psi_s, \psi_d)$ which computes the normalized ratio of this flow in both directions, and
- *Schema Coupling* $\chi(\Psi, \psi_i)$ of a schema which is the weighted measure of *Inter-Schema Coupling* of ψ_i and all n other schemas in Ψ.

4.4.4 Deterioration of Z Specifications

Ott and Thuss showed that slice-based measures are sensitive to changes and that they have the potential to quantify deterioration processes [39]. This sensitiveness was also demonstrated on sample Z specifications by Bollin [6], and the idea is now to make use of the measures to estimate the effect of maintenance actions applied to formal Z specifications.

The basic idea goes back to a simple perception: the fewer thoughts there are in one schema, the clearer and the sharper is the set of predicates. So, when a schema deals with many things in parallel, a lot of (self-contained) predicates are to be covered. This has an influence on the set of slices that are to be generated. When there is one "crisp" thought specified in the schema, then the slice intersection covers all the predicates. When there are different thoughts specified in it, then the intersection is expected to get smaller (as each slice only regards dependent primes). A progress towards a single thought should therefore appear as a convergence between the size of the schema and the size of its slice-intersection, an increasing divergence could indicate some deterioration of the formal specification.

Definition 16 Deterioration. *Let Ψ be a formal Z specification, ψ_i one schema (out of n schemas) in Ψ, and $SP^{fb}_{\cap}(\Psi, \psi_i)$ its slice intersection. Then Deterioration $\delta(\Psi)$ expresses the difference between the average schema size and the average size of the slice*

intersections. It is defined as follows:

$$\delta(\Psi) = \frac{\sum_{i=1}^{n} \mid \psi_i \mid - \mid SP_{\cap}^{fb}(\Psi, \psi_i) \cap \psi_i \mid}{n}$$

The value of deterioration, when standing alone, is not very informative – it is just an absolute number depending on the context. More information yields looking at the differences in the deterioration values between consecutive versions of the specification. For this case the notion of a *Relative Deterioration* is defined.

Definition 17 Relative Deterioration. *Let Ψ_{n-1} and Ψ_n be two consecutive versions of a formal Z specification Ψ. Then the relative deterioration ($\rho(\Psi_{n-1}, \Psi_n)$) (with n being the sequence number of specifications, $n > 1$) is calculated as the relative difference between the deterioration of Ψ_{n-1} and Ψ_n. It is defined as follows:*

$$\rho(\Psi_n) = 1 - \frac{\delta(\Psi_n)}{\delta(\Psi_{n-1})}$$

The value of relative deterioration is greater than zero when there is a convergence between schema size and slice intersection, and it is negative, when the differences between the sizes get bigger, indicating some probably unintentional deterioration.

4.5 The Study

After introducing the experimental basis and subjects in Sections 4.5.1 and 4.5.2, the study is split into three parts. In the first part (Section 4.5.3) a set of specifications is taken as basis in order to assess the usefulness of the measures. The objective is to find out whether each of the measures represents unique characteristics of the specification or not. In the second part (Section 4.5.4) a set of specifications, revisions of the Web Service Definition Language Specification (*WSDL*) 2.0, is assessed in a longitudinal study. The objective of this step is to find out how well the measures are suited for predicting deterioration effects. As a *CVS* system has been used for its further development from version 1.0, the 139 versions leading to *WSDL* 2.0 are a great candidate for examining the behavior of the measures. In the third part (Section 4.5.5) the results of the measures are examined again. The objective is to find out whether a baseline of measures can be found, and, if so, how a "typical" metrics distribution looks like.

4.5.1 Experimental Subjects

Large and publicly available specifications are rare. However, for the assessment of the measures a representative set of specifications is necessary. For this study several books, papers and projects have been searched and the resulting set of specifications has then been extended by "real-world" specifications like that of the above mentioned Web Service Definition Language. Tab. 4.5 provides a list of all specifications analyzed so far. The birthday book (*BB*) [47], *Elevator* [12], and *Access Control System* [28] specifications are classical paper and textbook examples. The *Petrol station*, *Cinema*, *Library* system, *BankAccount*, *Cycalize*, *VendingMachine*, and *ParcelService* specifications have been built by students during labs at Klagenfurt University. The rest of the specifications

Spec. Ψ	*Type*	*Lines*	$LOC(\Psi)$	*A4*	*Subj.*	*CC*	*Preds*
Library	S	43	37	2	2	33	13
Birthday Book	T	64	40	2	6	34	8
Petrol	S	130	88	3	10	65	34
Access Control	T	60	45	2	6	41	12
Cinema	S	175	95	4	9	74	26
OBS	R	377	106	7	13	104	108
Cycalize	S	245	118	7	10	96	30
Seguridad	R	278	130	4	18	124	56
VendingMachine	T	264	169	7	25	146	60
BankAccount	S	199	173	5	15	181	153
SchedulerJW	R	291	186	6	21	166	76
ParcelService	S	334	191	7	9	136	24
Elevator	T	241	193	6	18	185	264
SteamBoiler	T	331	252	9	36	342	181
EngineCache	R	384	259	8	34	225	86
CISC	R	746	343	12	67	368	120
ICLDataDic	R	945	352	20	30	278	173
ICT WM	R	539	359	12	41	280	154
OSScheduler	T	423	365	12	47	455	204
RoomPlanner	T	643	381	13	19	254	111
Scheduler	R	650	394	12	35	330	218
Caviar	R	1,500	456	27	92	467	189
Unix	R	1,199	474	29	58	362	112
EOCP	R	914	488	11	53	538	322
Hysteresis	R	6,594	604	93	19	303	69
Teleservice	R	954	679	12	40	686	338
CVSServer	T	1,515	766	34	37	425	135
Telecommand	R	1,489	963	22	78	1,073	491
Radiation Therapy	R	2,482	1,172	36	131	1,038	704
WSDL 2.0	R	16,649	814	33	98	814	3,784
Mondex	R	4,857	1,371	102	132	868	1,535
Corba	R	3,370	1,616	81	130	1,268	425
FlowxSystem	R	6,128	1,941	94	284	2,129	2,148
Tokeneer	R	6,742	2,013	117	170	1,575	2,141
ECSE7041	R	3,293	2,063	73	134	1,461	823
Sum		**65,190**	**20,193**	**924**	**1,938**	**17,040**	**15,633**
WSDL (47 vers.)	R	571,559	37,348	-	2,978	3,076	85,941
Total		**636,749**	**57,541**	**-**	**4,916**	**20,116**	**101,574**

Table 4.5 Sample specifications used in this study. The table summarizes the type (T .. Textbook, S .. Students solution, R .. 'Real-world' specification), scale (total lines, lines of specification text, size in A4 pages), the number of schemas (*Subj.*), its conceptual complexity (*CC*), and the number of predicates (*Preds*) that have formed the abstraction criteria.

are: sample specifications from the test case framework called "Fastest" [16] (OBS – On-Board Application Software System, $Seguridad$, $EngineCache\ Scheduler$, $Teleservice$, $EOCP$ – the EXP-OBDH Communication Protocol, $Telecommand$, $FlowxSystem$, and $ECSSE7041$ – the ECSS-E-70-41A Standard), the $ICT\ Window\ Manager$ [7] which is part of the scheduler for the Linux 2.0 kernel, the $Radiation\ Therapy\ Machine$ [29], the $Mondex$ electronic purse [53], the $Caviar\ System$ [25, pp.71–98], the $Hysteresis$ specification [2], the $ICL\ Data\ Dictionary$ [25, pp.99–119], the $OS\ Scheduler$ [52, pp.330-344], the $CISC\ Temporary\ Storage$ [25, pp.203–218], and the $Unix\ File\ System$ [25, pp.41–70]. The $Steamboiler$ ([1, pp.109–128]), $RoomPlanner$, $CVSServer$, and $Corba$ specifications have been taken from the example set of the Isabelle/HOL-Z project version 3.0 (beta) [10], and the $Scheduler$ specification has been taken from the test-bench of the CZT project [21]. The $Tokoneer$ specification was created by Praxis and $SPRE$ as a demonstration object for the NSA [15].

For conducting this study, a Java-Z framework called ViZ [5] was used. ViZ supports concept location within formal Z specifications. It takes a formal Z specification as input by making use of the CZT parser for Z specifications [36], transforms it to an $ASRN$, and then calculates the necessary slices. For this study it has been equipped with batch-functionality to access the 139 versions of the $WSDL$ specification and to export the measures into a comma-separated file for further statistical processing. The only manual intervention necessary was adapting a few lines in every $WSDL$ specifications to match the actual Z standard as implemented in the CZT plug-in.

The $WSDL$ specification is available in 139 versions in the CVS, but most of the revisions vary in the amount of comments and natural text describing the specification only. 47 of them differ in the specification code itself and therefore they become perfect candidates for this study. The 47 versions result in 37,348 lines of specification text and are the basis for 85,941 slices. In total 4,916 Z schemas and 57,541 lines of specification text have been analyzed, summing up to more than 101,000 predicates, slices have been calculated for.

4.5.2 Statistics

Within the scope of this contribution three different statistical tests were used to assess the data: the Pearson's Correlation Coefficient, the Spearman's Rank Correlation Coefficient, and Kendall's Tau Correlation Coefficient.

The Pearson's correlation coefficient (R_P) measures the degree of association between the variables, assuming normal distribution of the values [42, p. 212]. Though this test might not necessarily fail when the data is not normally distributed, the Pearson's test only looks for a linear correlation. It might indicate no correlation even if the data is correlated in a non-linear manner.

As past experiences about the distribution of the values are missing, one has to assume the data being not normally distributed. To handle this case the Spearman's rank correlation coefficient (R_S) has been chosen [42, p. 219]. It is a non-parametric test of correlation and assesses how well a monotonic function describes the association between the variables. This is done by ranking the sample data separately for each variable.

Kendall's robust correlation coefficient (R_K) can be used as an alternative to the Spearman's test [20, p. 200]. It is also non-parametric and investigates the relationship among pairs of data. However, it ranks the data relatively and is able to identify partial correlations.

When there is no likelihood of confusion, then R will be used to refer to either R_P, R_S or R_K.

In the remainder of this work the correlation R is interpreted as follows:

- When $\mid R \mid \in [0.8,\ 1.0]$ it is interpreted to indicate a *strong association*.
- When $\mid R \mid \in [0.5,\ 0.8)$ it is interpreted to indicate a *moderate association*.
- When $\mid R \mid \in [0.0,\ 0.5)$ it is interpreted to indicate a *weak association*.

In addition to the values of the correlation R, also the significance level (p) of the value is provided (checking, within the scope of the null hypothesis, that the probability of the value of R is bigger or equal to the observed value of R). The values in the following tables are rounded to the third decimal place (which means that a value of $p = 0.0005$ would become $p = 0.001$).

4.5.3 Comparison of Metrics

A first step is to take a closer look at the evaluation of the measures with respect to their uniqueness. This includes the following issues to be checked:

I1 the relationship between size and structure based measures (*Conceptual Complexity*, *Logical Complexity* and *Definition-Use count*) and the measure of *Lines of Specification Text*,

I2 the relationship between slice-based measures (*Tightness*, *Coverage*, *Overlap* and *Coupling*) and one size-based measure (*Conceptual Complexity*), and finally,

I3 the correlations between different slice-based measures (*Tightness*, *Coverage*, *Overlap*, and *Coupling*).

4.5.3.1 Issue I1. The first issue is to look at the relationship between the size and structure based measures and the measure of lines of specification text. As the specification is mapped to the $ASRN$ one-to-one (and the measures are all calculated on the basis of the graph), it might be possible that especially the previously defined complexity measures are just proxies for the number of lines in the text. The $ASRN$ contains a vertex for every prime, and theoretically, between each pair of vertices representing predicates there might be a control and/or data dependency.

Figure 4.2 presents a scatter plot that compares the lines of specification text on the X-axis (per Z schema ψ_i) and related size- and structure-based measures.

As might have been expected, there is a clear trend for the size-based measure (conceptual complexity CC, upper left), indicating a high correlation. But the plot is not so clear for structure based measures. The bigger the specifications, the more dependencies will usually exist, but the plots in figure 4.2 demonstrate that the plot broadens for both complexity measures – they are not strongly related to the size of the specification.

Table 4.6 summarizes the results of the correlation tests. As expected, CC is strongly related to the number of lines of specification text ($R = 0.933$ for the Pearson test). However, the two other measures (logical complexity and Def/Use count) do have moderate

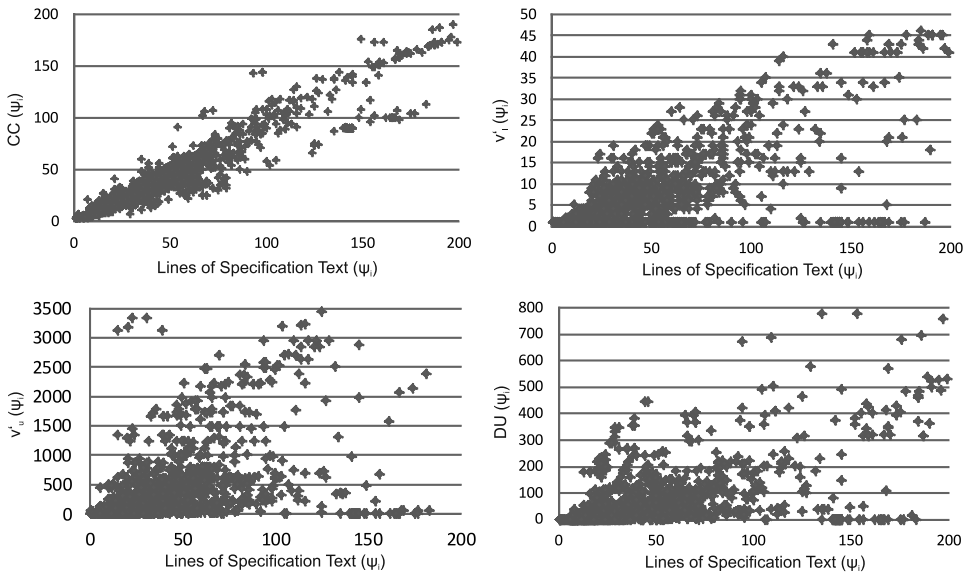


Figure 4.2 Metrics comparison of size and structure-based measures (*Conceptual Complexity*, *Logical Complexity* and *Definition-Use Count*) against the measure of *Lines of Specification Text* (n=1938).

Lines of Specification Text Correlation (n=1938)							
		Pearson		Spearman		Kendall	
Sig.	*Measure*	R_P	p	R_S	p	R_K	p
Strong	$CC(\Psi)$	0.933	.000	0.964	.000	0.858	.000
Moderate	$v_l'(\Psi)$	0.656	.000	0.563	.000	0.470	.000
[0.5, 0.8)	$v_u'(\Psi)$	0.668	.000	0.580	.000	0.458	.000
	$DU(\Psi)$	0.659	.000	0.557	.000	0.435	.000

Table 4.6 Pearson's, Spearman's and Kendall's correlation (R_P, R_S, R_K including p-values for testing the hypothesis of no correlation against the alternative that there is a nonzero correlation) for the metrics comparison displayed as scatter plot in Fig. 4.2. The classification of the statistical significance (*Sig.*) is based on the Pearson correlation values R_P.

correlations only. As all three tests yield a (weak to) moderate relation, the structural measures are very likely not just proxies for the number of specification lines.

The correlation tests (and the scatter plot) indicate that CC does not provide new insights when compared to the lines of specification text ($LOC(\Psi)$). Nevertheless, the use of CC in the remainder of this work can be justified by the following three arguments:

- The number of lines of specification text depends on the style used when writing down the predicates. By looking at the differences between the values of CC and lines of specification text in Table 4.5, one can see that the values differ up to 25 percent – which is quite a lot.
- CC represents the amount of declarations and predicates in a specification (which are very likely refined later on to program code). So, CC might be slightly more precise as basis for implementation-related predictions.
- The calculation of the other semantics-based values goes back to counting primes. When there are correlations, then it is very likely that the effect will be more pronounced when looking at primes and not at the number of lines of specification text.

4.5.3.2 Issue I2. The second issue is to take a closer look at the correlation between the size of the specification (now represented by $CC(\psi_i)$) and the values of cohesion and coupling. Figure 4.3 presents a scatter plot for the six different measures. None of them indicates a direct or strong correlation, but, the plots do reveal other interesting properties.

- It is apparent that the measures for coupling and cohesion do have their bulks in different areas of the diagrams. The values for cohesion are in most cases between 0.5 and 1.0, whereas the values for *Coupling* are between 0.0 and 0.5. Fig. 4.4 illustrates this for the values of *Coupling* and *Coverage*. As explained later, there is some weak to moderate relation between them. It is good to see that also for formal specifications there seems to be some trend towards values of high cohesion and low coupling.
- Four of the five values of cohesion (τ, Cov_{min}, Cov_{max}, O) do have concentrations of the values at 0 and 1. The reasons go back to the density and the types of the specifications. There are specifications that contain unrelated state spaces without predicates. The values for cohesion are then equal to 0. And some of the specifications are very dense (with relations between all its primes). Every prime is then element of the intersection slices. The values for cohesion are thus equal to 1. Basically, these situations happen with text-book examples (which try to explain the syntax of Z in a condensed manner), but also in the $Caviar$ and the $Unix$-system specification. In these specifications up to 24% of at least one of the values are equal to 0 or 1. However, in 40% of these cases not all of the values of *Coupling* do have the the same value then, and an assessment of these schemas is still possible.

Apart from these two observations, the plots between 0 and 1 do not reveal a clear correlation. Table 4.7 summarizes the results of the Pearson, Spearman, and Kendall tests. The tests show that with raising sizes of the specifications the values for cohesion are decreasing, whereas the value for *Coupling* increases. This is not surprising, as with larger specifications the chance for relationships between *all* of the schemas in a specification decreases. The correlation values of the Spearman test are slightly higher than the values of the other two tests, indicating a tendency to a non-linear correlation. However, the results are still in the weak association class and thus the measures are treated as basically unrelated in the remainder of this work.

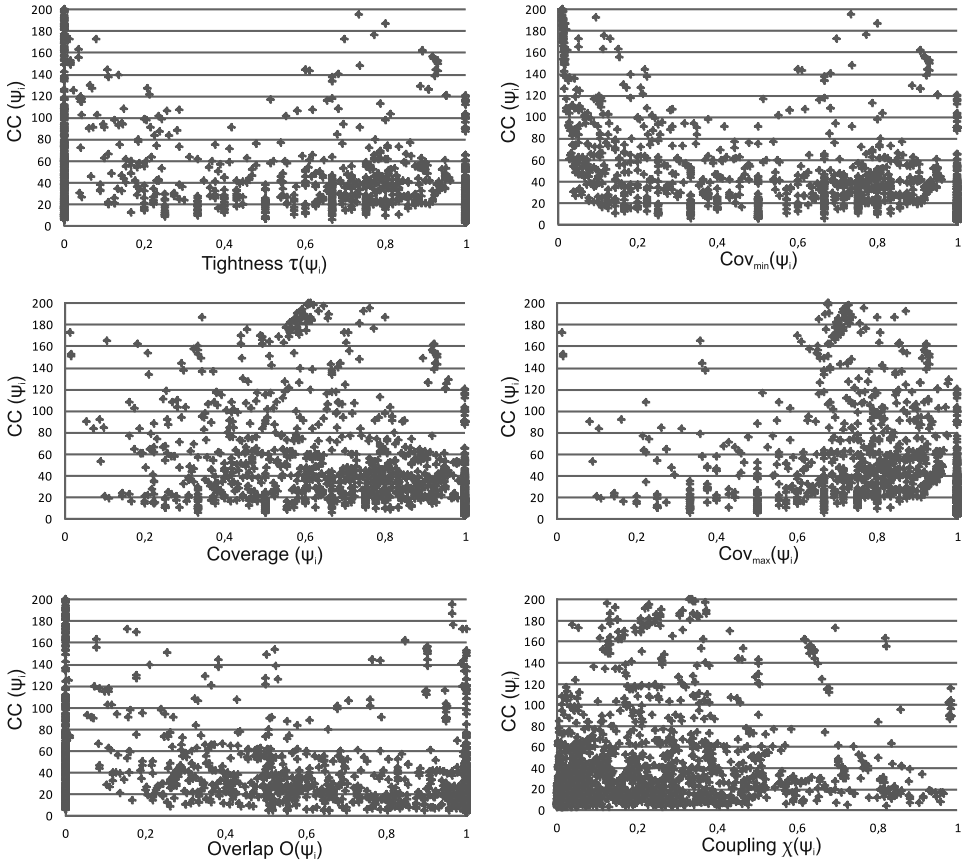


Figure 4.3 Metrics comparison between size-based and slice-based measures (n=1938).

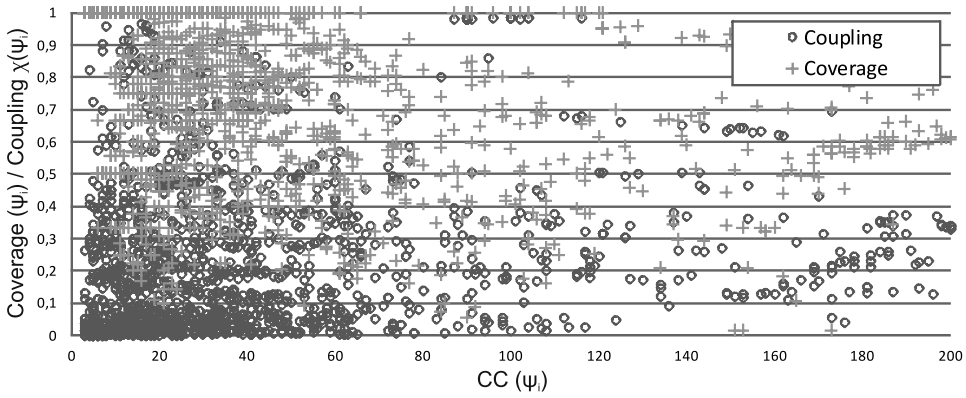


Figure 4.4 Distribution of two slice-based measures (*Coupling* and *Coverage*) for all samples under test (n=1938).

Conceptual Complexity Correlations (n=1938)							
		Pearson		Spearman		Kendall	
Sig.	*Measure*	R_P	p	R_S	p	R_K	p
Weak	*Tightness*	-.296	.000	-.436	.000	-.330	.000
[0.0, 0.5)	Cov_{min}	-.360	.000	-.498	.000	-.378	.000
	Coverage	-.221	.000	-.387	.000	-.281	.000
	Cov_{max}	-.103	.000	-.308	.000	-.226	.000
	Overlap	-.314	.000	-.445	.000	-.331	.000
	Coupling	0.234	.000	0.311	.000	0.211	.000

Table 4.7 Pearson's, Spearman's and Kendall's correlation between *CC* and all slice-based measures displayed as scatter plot in Fig. 4.3 (including p-values for testing the hypothesis of no correlation against the alternative that there is a nonzero correlation). The classification of the statistical significance (*Sig.*) is based on the Pearson correlation values.

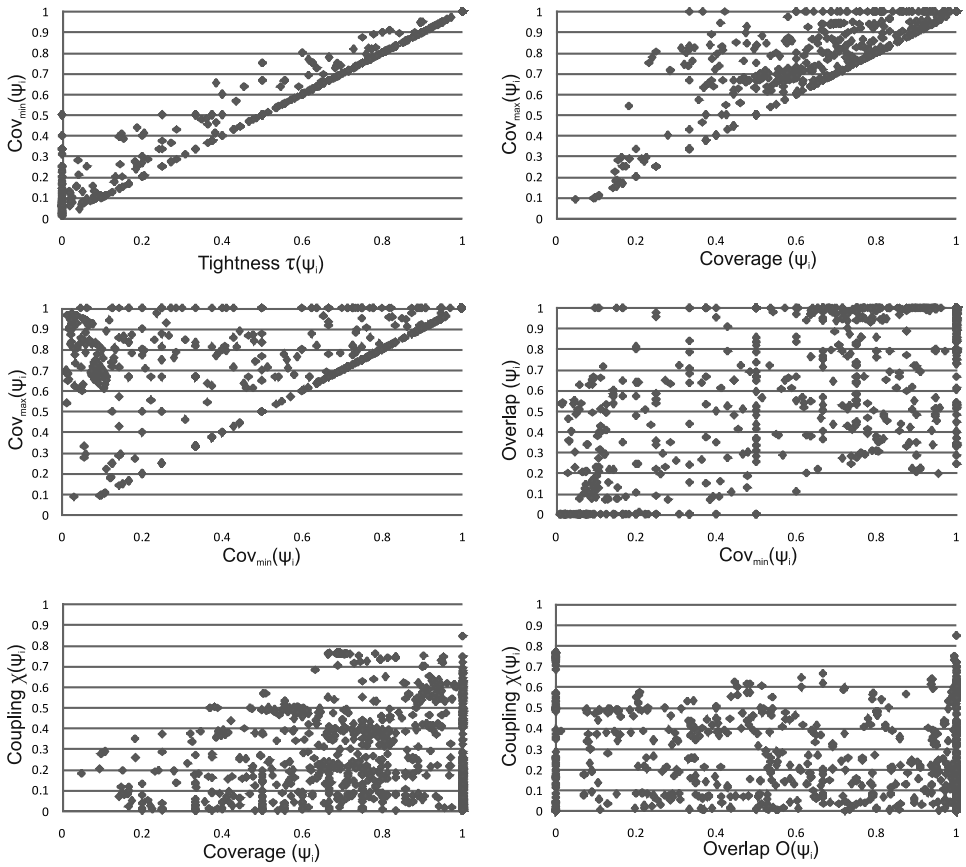


Figure 4.5 Metrics comparison between slice and structure-based measures (n=1938).

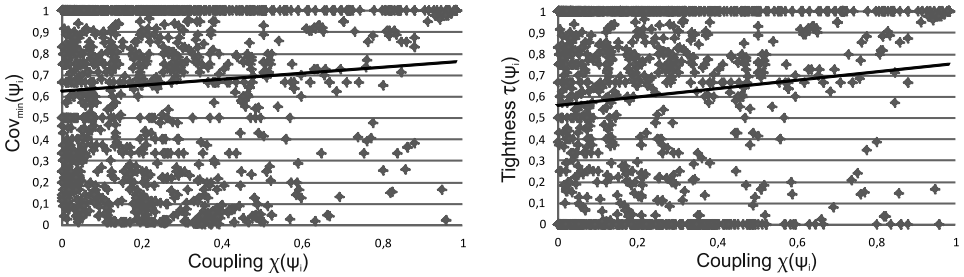


Figure 4.6 Metrics comparison between slice-based measures against the value of *Coupling* including their trend-lines (n=1938).

Slice-based Measures Correlations (n=1938)								
Sig.			Pearson		Spearman		Kendall	
	Measure 1	*Measure 2*	R_P	p	R_S	p	R_K	p
Strong	*Tightness*	Cov_{min}	0.975	.000	0.981	.000	0.924	.000
[0.8, 1.0]	*Tightness*	*Coverage*	0.891	.000	0.947	.000	0.839	.000
	Cov_{min}	*Coverage*	0.881	.000	0.937	.000	0.812	.000
	Coverage	Cov_{max}	0.831	.000	0.852	.000	0.738	.000
	Tightness	*Overlap*	0.808	.000	0.726	.000	0.596	.000
Moderate	Cov_{min}	*Overlap*	0.779	.000	0.700	.000	0.554	.000
[0.5, 0.8)	*Tightness*	Cov_{max}	0.648	.000	0.760	.000	0.596	.000
	Coverage	*Overlap*	0.638	.000	0.607	.000	0.476	.000
	Cov_{min}	Cov_{max}	0.624	.000	0.754	.000	0.613	.000
Weak	Cov_{max}	*Overlap*	0.358	.000	0.368	.000	0.287	.000
[0.0, 0.5)	*Coupling*	*Coverage*	0.208	.000	0.080	.000	0.064	.000
	Coupling	Cov_{max}	0.203	.000	0.084	.000	0.065	.000
	Coupling	*Tightness*	0.125	.000	0.016	**.477**	0.013	**.412**
	Coupling	Cov_{min}	0.101	.000	-.024	**.290**	-.017	**.298**
	Coupling	*Overlap*	0.077	.001	-.094	.000	-.061	.000

Table 4.8 Pearson, Spearman and Kendall for the correlation of slice-based measures (including p-values for testing the hypothesis of no correlation against the alternative that there is a nonzero correlation). The classification of the statistical significance (*Sig.*) is based on the Pearson correlation values.

4.5.3.3 Issue I3. The last issue in evaluating the expressiveness of the values is to look at the interrelations between and within structure and slice-based measures. Figure 4.5 presents some scatter plots for the values. The results of the correlation tests are summarized in tables 4.8 (for the slice-based measures) and 4.9 (for structure and semantics-based measures).

The correlation values for the slice-based measures (Table 4.8) differ widely. Especially the value of *Tightness* is highly correlated to the other measures, and according to the test results, it is very likely that *Tightness* is a proxy for some of the other cohesion-related

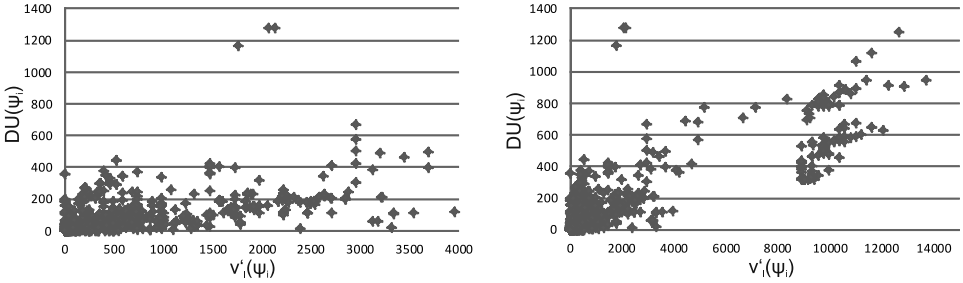


Figure 4.7 Scatter plots for the Def/Use count value and logical complexity v_l'. On the left, there is the plot for the values of $v_l' \leq 4000$ (n=1650), and to the right the full plot for the values of $v_l' \leq 14000$ (n=1932).

measures. In fact, the definitions of *Tightness*, *Coverage* and *Overlap* are quite similar. Not surprisingly, this relation can also be seen in Fig. 4.5. The least correlated measures are *Coupling* and *Overlap*, and they thus are excellent candidates for the measures we need in the remainder of this work.

All p-scores are less than or around 0.001 – with two exceptions: firstly, at the correlation between $Coupling$ and $Tightness$ where p is higher than $0,05$ (0.477 for the Spearman, and 0.412 for the Kendall tests), and secondly, at the correlation between *Coupling* and Cov_{min} (where p is 0.29 for the Spearman test and 0.298 for the Kendall test). This indicates that the results for these two tests are not statistically significant. A closer look at the scatter plot in Figure 4.6 does not explain the reason(s) for this situation. The plot seems to confirm the result of the Pearson correlation test, which tell us that there is, if at all, just a slightly positive relation.

But, a closer look into the core data reveals the matter: in Z specifications, operation schemas are usually combinto form bigger operations, finally (hierarchically) covering all the operations on the states. These operations (about 10% of all the schemas) do not add any new predicates or declarations, but just include all other existing operations and states. They are really big and (due to the introduced redundancy) distort the calculation of the correlation values. Omitting them would basically lead to the same correlation values, but with p-scores below 0.005. During the design of the experiments it was decided not to prune the set of specification schemas. As combined operation schemas are typical for Z specifications, such an omission of schemas would have falsified the set of experimental subjects.

The situation is clearer for the combinations of structure-based and slice-based measures. Table 4.9 shows that there is only a weak association between structure-based and slice-based measures. Table 4.10 reveals that there is a strong correlation between the set of structure-based measures. However, concerning these tables, two things are particularly noticeable.

At first, all p-scores in both tables are around 0.0 – with one exception: the p-score for the correlation test between $v_l'(\psi)$ and $Coupling$ in Table 4.9 is 0.337 for the Spearman test and 0.146 for the Kendall test. Secondly, the Pearson, Spearman and Kendall correlation values for the structure-based measures diverge more than in the tests before. Again, a closer look at the data reveals the reason: 41 (2.11%) of all schemas are enormously big and do contain $10,000$ and more control dependencies, whereas 75% of the schemas contain less than 500 control dependencies. The rest of the schemas has between 500 and

Size- and Slice-based Measures Correlations (n=1938)								
			Pearson		Spearman		Kendall	
Sig.	*Measure 1*	*Measure 2*	R_P	p	R_S	p	R_K	p
Weak	$v_l'(\psi)$	*Tightness*	-.341	.000	-.301	.000	-.241	.000
[0.0, 0.5)	$v_l'(\psi)$	Cov_{min}	-.414	.000	-.356	.000	-.279	.000
	$v_l'(\psi)$	*Coverage*	-.257	.000	-.287	.000	-.221	.000
	$v_l'(\psi)$	Cov_{max}	-.155	.000	-.239	.000	-.193	.000
	$v_l'(\psi)$	*Overlap*	-.368	.000	-.204	.000	-.235	.000
	$v_l'(\psi)$	*Coupling*	0.022	**.337**	0.033	**.146**	0.051	.002
	$v_u'(\psi)$	*Tightness*	-.362	.000	-.360	.000	-.275	.000
	$v_u'(\psi)$	Cov_{min}	-.438	.000	-.430	.000	-.322	.000
	$v_u'(\psi)$	*Coverage*	-.265	.000	-.315	.000	-.236	.000
	$v_u'(\psi)$	Cov_{max}	-.171	.000	-.275	.000	-.210	.000
	$v_u'(\psi)$	*Overlap*	-.361	.000	-.306	.000	-.231	.000
	$v_u'(\psi)$	*Coupling*	0.054	.017	0.307	.000	0.220	.000
	$DU(\psi)$	*Tightness*	-.343	.000	-.355	.000	-.270	.000
	$DU(\psi)$	Cov_{min}	-.411	.000	-.421	.000	-.315	.000
	$DU(\psi)$	*Coverage*	-.252	.000	-.303	.000	-.226	.000
	$DU(\psi)$	Cov_{max}	-.120	.000	-.225	.000	-.173	.000
	$DU(\psi)$	*Overlap*	-.375	.000	-.342	.000	-.258	.000
	$DU(\psi)$	*Coupling*	0.152	.000	0.409	.000	0.311	.000

Table 4.9 Pearson, Spearman and Kendall for the correlation of slice-based and sized-based measures (including p-values for testing the hypothesis of no correlation against the alternative that there is a nonzero correlation). The classification of the statistical significance (*Sig.*) is based on the Pearson correlation values.

Structure-based Measures Correlations (n=1938)								
			Pearson		Spearman		Kendall	
Sig.	*Measure 1*	*Measure 2*	R_P	p	R_S	p	R_K	p
Strong	$v_u'(\psi)$	$DU(\psi)$	0.918	.000	0.813	.000	0.660	.000
[0.8, 1.0]	$v_l'(\psi)$	$v_u'(\psi)$	0.871	.000	0.648	.000	0.531	.000
	$v_l'(\psi)$	$DU(\psi)$	0.834	.000	0.644	.000	0.518	.000

Table 4.10 Pearson, Spearman and Kendall for the correlation of slice-based and sized-based measures (including p-values for testing the hypothesis of no correlation against the alternative that there is a nonzero correlation). The classification of the statistical significance is based on the Pearson correlation values.

4000 control dependencies. Figure 4.7 visualizes the difference by making use of different intervals: at the left there is the scatter plot and the trend-line for schemas of "average size" (n=1650), and to the right there is the scatter plot for the he values of $v_l' \leq 14000$ (n=1932). As the Spearman test is based on the rank order of the values, it is not sensitive to the difference of an order of magnitude – which also explains the difference in the R values. The correlation values for the pruned set of schemas would all be between 0.648 and 0.791, so indicating a moderate correlation only.

For describing the complexity of a specification's structure it could be sufficient to take just a look at either one value of logical complexity or the Def/Use count. However, for schemas of "normal" size, all the values could be relevant as there is only moderate correlation between them. Additionally, the absolute values provide a deep insight into the specification (and a possibly refined implementation).

To summarize, when looking for a set of measures that represents unique characteristics of a specification (size, structure, semantics) then the outcome of this part of the study suggests to focus on the values of

- *Conceptual Complexity* (CC) or *Lines of Specification Text* when talking about its size,
- *Logical Complexity* ($v'(l,u)$) or *Definition-Use Count* (DU) for its structural complexity,
- *Coupling* (Ξ), *Overlap* (O), and eventually *Coverage* (Cov) for its semantic characteristics.

The measure of *Tightness* is skipped due to its ambiguity of its correlation results (p-scores) and due to its high correlation to the other cohesion measures. *Minimum* and *Maximum Coverage* are skipped for the same reasons. On the other hand, the remaining measures (CC, $v'(l,u)$, DU, Ξ, O and Cov) will be used in the next two sections of this paper.

4.5.4 Longitudinal Study

The Web Service Definition Language [13] is one of the rare, big publicly available, formal Z specifications. It specifies the Web Services Description Language Version 2.0 ($WSDL$ 2.0) which is an XML language for describing Web services. The specification consists of a detailed natural language description and a Z specification of the core language that is used to specify Web services based on an abstract model of what the service offers. Additionally, it specifies the conformance criteria for documents in this language.

The specification itself is of medium complexity. There are no predicates that are mathematically very complex. 78% of the expressions make use of the logical AND operator, and 17% make use of existential quantifiers. Only 1.5% make use of logical implications. The final revision contains 814 primes (distributed over 1,413 lines of Z code, in 16,641 lines of text altogether). The advantage of this specification is that, starting with 2004 (and during the development of $WSDL$ 2.0) a concurrent versioning system (CVS) has been used. $WSDL$ 1.0 is not available in Z, but from November 2004 on 139 versions - due to change requests and other maintenance activities - have been checked in till its final release in 2007 [44]. The first revision is an adoption of $WSDL - 1.0$, and then, successively, functionality has been added, modified, or deleted. Fig. 4.8 presents an overview of the changes in the size of the specification (comments and specification text) for all 139

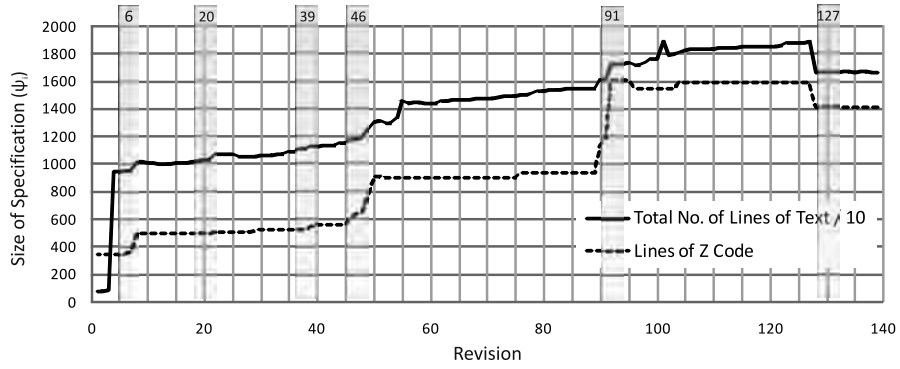


Figure 4.8 Number of lines of specification text and comment lines for all 139 revisions of *WSDL* 2.0. The revisions that are discussed in more detail in the study are highlighted.

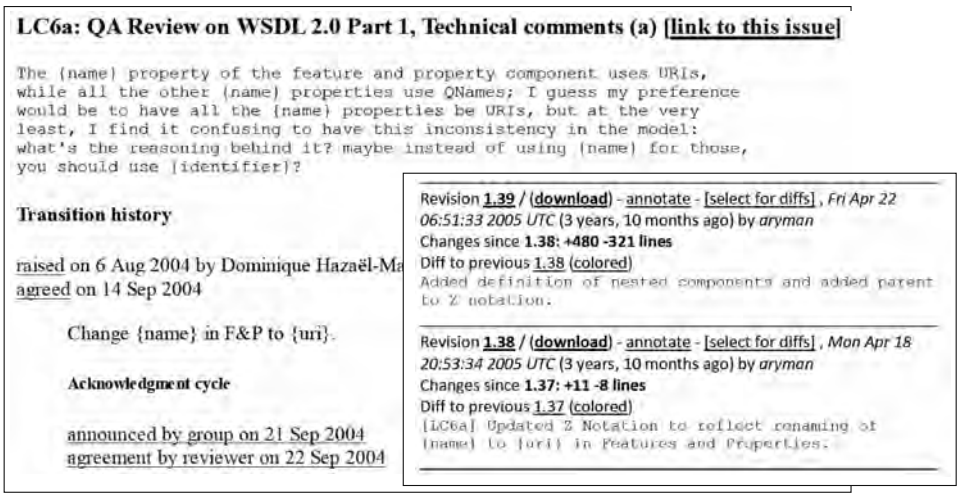


Figure 4.9 Example of a "Last Call" description ($[LC6a]$) and two entries (Revs. 1.038 and 1.039) in the CVS log for the *WSDL* 2.0 specification.

versions. As can be seen, the first version in the CVS (with revision number 1.1) contained 348 lines of specification text (790 lines of text in total). The specification has steadily been extended, finally yielding a specification (with revision number 1.139) consisting of 1.413 lines of specification text. The last revision in the CVS became the publicly available Z specification of *WSDL* 2.0. Please note that for reasons of space the figures' revisions 1.1 to 1.139 are abbreviated to 1 to 139.

Before taking a closer look at the different measures of the *WSDL* specification, a look at the CVS-log is very informative. It provides a first insight into the types of changes (corrective, adaptive, or perfective) that occurred on the way to the final release. Up to revision 1.092 the interventions (called "last calls") mainly consist of adding and modifying concepts (schemas) to the specification (see Fig. 4.9 for an example, where, at revision 1.038 the last call [LC06a] has been implemented). After revision 1.095 there are solely

change requests. Though there have been several changes influencing the final product, the following sections and revisions attract attention and are considered later in more detail:

- *Up to Rev. 1.006* there is a mapping of $WSDL$ version 1.0 to Z. Only minor changes to the specification take place, but, starting with Rev. 1.007 editorial improvements take place. After the improvements, the model is extended (for example by introducing the two concepts of $Binding$ and $Services$).

- Starting with *Rev. 1.020* a recurring refactoring and extension of the model takes place. Several concepts are added, for example the notion of an $Interface\ Component$ or that of an ID).

- Starting with *Rev. 1.038* another simplification takes place. The concepts of $URIs$ and $QNAME$s are merged.

- Beginning with *Rev. 1.045* a lot of constraints are added to the specification. Several smaller changes are applied to the model (according to $Interface$ and $Bindings$). Additionally, $Faults$ are introduced as a stand-alone concept.

- At *Rev. 1.077* editorial improvements to message label rules take place. In addition to that long definitions are refactored in the $tables$ part.

- Beginning with *Rev. 1.090* the model is extended massively. New concepts, like that of an $XMLnamespace$ or $TypeDesignator$s are introduced.

- At *Rev. 1.096*, according to the CVS log, a simplification of the model takes place. In fact, $Component$s and their $Context$ are aggregated into a $TopLevel\ Component$, additionally, the concept of IRIs is updated.

- Beginning with *Rev. 1.127*, change requests lead to a structural refactoring of the specification. Several concepts, like that of $Features$ and $Properties$ are removed.

Massive changes took place at revisions 1.039ff and 1.090ff. It has to be noted that, after looking at the CVS-logs, the changes have been verified by making use of a software tool for file-compare (Beyond Compare V3.2.4) which confirmed the entries in the log. By analyzing the different maintenance activities stored in the CVS-log, it is noticeable that a lot of the last calls followed a clear strategy in adding the desired functionality. The steps are as follows:

1. Refactoring the actual version.

2. Adding/removing/modification of a concept.

3. Update of the natural language documentation.

This strategy can be derived from the CVS-log directly. The next question addressed is now whether the measures that have been defined in Sec. 4.4 capture these changes, and whether they can be used to assess the results of the different maintenance activities.

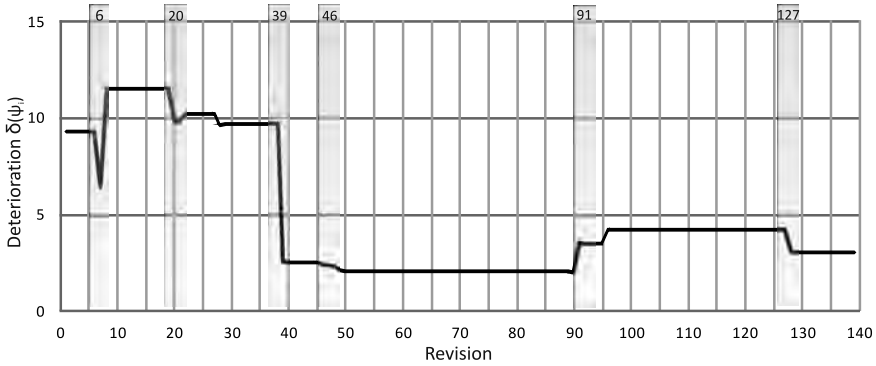


Figure 4.10 Values of deterioration for all 139 revisions of the *WSDL* specification.

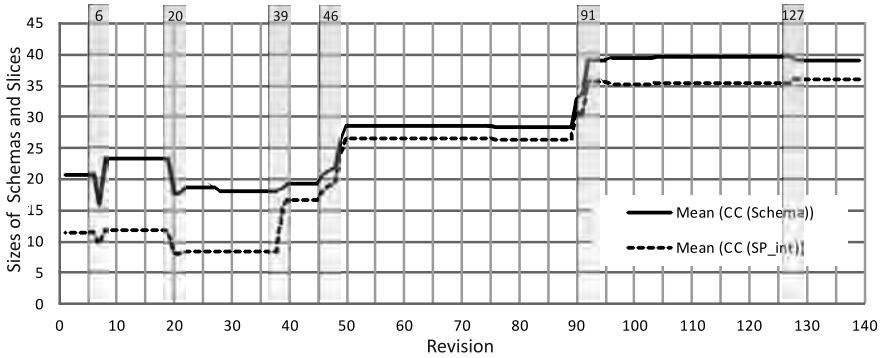


Figure 4.11 Change in the sizes of schemas and their corresponding intersection slices SP^{fb}_{cap} for all 139 revisions of the *WSDL* specification.

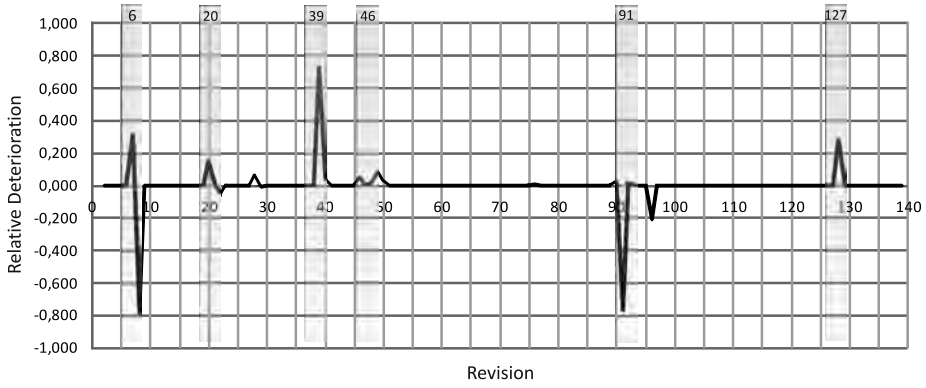


Figure 4.12 Relative change of the size of deterioration for all 139 revisions of the *WSDL* specification.

4.5.4.1 Deterioration At first, let us look at the measure called deterioration δ. Fig. 4.10 presents the value for δ for all 139 revisions in the CVS. This figure suggests that the specification deteriorates basically around revisions 1.006 and 1.091, and improves around revisions 1.039 and 1.127. In fact, the CVS-log also documents the changes, as they mention the extensions to the existing model after revisions 1.006 and 1.090, and also the simplifications at revisions 1.038 and 1.127.

At all revisions Z-code has been added or removed, but the change in the total size of the specification is not the reason for deterioration. Fig. 4.11 shows the values for the mean of the sizes of the Z schemas and the mean of the size of the corresponding slice intersections. Deterioration is the difference between the (average) size of a schema and the (average) size of the slice intersection (see Figure 4.11), thus by the drift between them we can see if and when the specification *semantically* deteriorates. It is noticeable that not at all revisions a change in size led to the same amount of change in the size of the slice intersections. For example, at revision 1.039 there is growth in size and a bigger growth in $SP^{b}_{\cap}$. This indicates that currently available concepts have been changed; existing trains-of-thought have been merged. On the contrary, at revision 1.091, there is an increase in size and a lower increase in $SP^{fb}_{\cap}$. This indicates that such primes have been added to the specification schemas that introduce new thoughts. The CVS log indeed confirms this observation: at revision 1.091 the new concepts of $XMLNamespaces$ and $TypeDesignator$ have been introduced, whereas at revision 1.039 existing schemas have been modified by simplifying the access to the $names$ concept.

4.5.4.2 Relative Deterioration The absolute values make it difficult to estimate how big the influence of a change is. For this reason the notion of relative deterioration (ρ) has been introduced in Sec. 4.4. Fig. 4.12 presents the value of ρ for all 139 revisions. Positive values indicate that the difference between the schema sizes and their slice intersections is reduced; this is assumed to be positive with respect to deterioration. On the other hand, negative values indicate negative effects.

When looking at Fig. 4.11, it is hard to say whether the difference between the sizes of the schemas and their slice intersections changes drastically or not. As the amount of the change (in respect to the size of the schema) is not made explicit, it only can be assumed that the most influencing modification happened at revision 1.039. With the notion of the relative deterioration the degree of the change becomes more visible. From Fig. 4.12 we

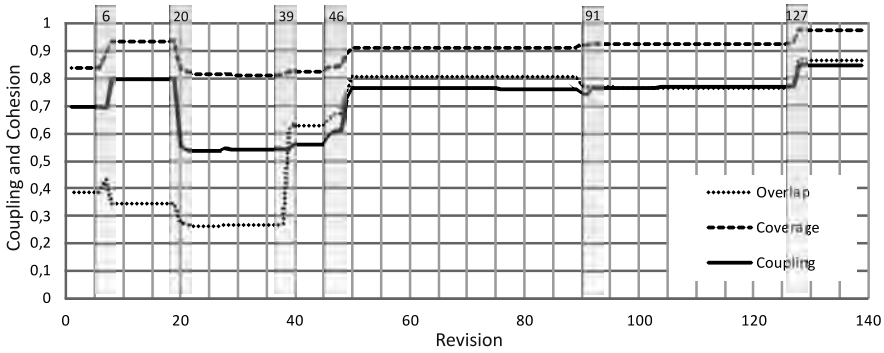


Figure 4.13 Values of *Coupling*, *Overlap* and *Coverage* for all 139 revisions of the *WSDL* specification.

can additionally infer that the biggest negative changes happened between revisions 1.007 and 1.008 and between revisions 1.090 and 1.091. Besides the improvement at revision 1.039, the final refactoring activities around revision 1.127 had also a very positive effect with respect to the value of deterioration.

Another interesting observation can be made: when taking a closer look at Fig. 4.12, the process of implementing last calls gets visible. At revisions 1.006, 1.020, 1.027 and 1.038 it can be deduced from the graph. A change is triggered, first, by a structural improvement (to be seen as a positive amplitude of ρ in the diagram), followed by the introduction of the new thought which in some of the cases resulted in a negative amplitude of ρ.

4.5.4.3 Coupling and Cohesion Up to now we have only looked at the differences between slice intersections and schema sizes. This section deals with the question of how the changes affect our previously defined quality measures.

Looking at Fig. 4.13, we see that the values for *Overlap* are low at the beginning (around 0.39). A lot of concepts within the specification schemas are unrelated. Even more, the number of primes common to all other slices got lower by adding new concepts into the specification schemas. The drop in the value of *Overlap* can be seen in Fig. 4.13 between revisions 1.006 and 1.038. Only with the combination of the concepts of $names$ at revision 1.038 this trend stopped and *Overlap* increased again (and reaches the value of 0.86 at the end).

The value of *Coverage* follows the fluctuation of the other measures – but not at all revisions to the same extent. On the long run, it increases, ending up at a slightly higher value than at the first revision. *Coverage* tells us about how specific a schema is (which means how many primes in its body are responsible for a single thought). The developers started at a high level but they managed to improve this property till the final release. It is interesting that the extension at revision 1.091 did not lead to a drastic decline of the measure. The reason for that is the fact that, with the introduction of new concepts a lot of crisp schemas were added to the existing specification. The changes on the existing schemas were kept marginal, resulting in only minor fluctuations of the value on average.

Coming from intra-schema measures to inter-schema properties, *Coupling* refers to the flow between different schemas in the specification. The less the flow, the better. In case of

$WSDL$, this flow is quite high. Fig. 4.13 shows that the value fluctuates with the values of *Coverage* and *Overlap*, but again not necessarily to the same extent, and not inversely (as would be assumed to be ideal for programs). Beginning with version 1.006, the model has been changed in such a way that schemas started to share concepts, yielding an increase in *Coupling*. Then, at revision 1.020 new, self-contained, schemas were added and the value decreased again. After that, at revision 1.046, a new schema with connections to existing schemas was introduced. The result is an increase in the value of *Coupling*. Interesting is also the situation around revision 1.090. There, massive changes to the specification happened, but they barely led to a change of this quality measure. In fact, the freshly added schemas are as highly interconnected with the other schemas, as the schemas that existed before. Finally, at revision 1.127 the specification has been refactored a little. Some concepts have been removed; others were integrated into single schemas. *Coupling* increases as the number of relations between the concepts has been increased a slightly.

To summarize, the value of *Coupling* and the different cohesion values "visualize" a lot of changes documented in the CVS log. But they are not necessarily able to evaluate every change. One reason for this is that the measures are not sensitive enough when only minor changes take place (and the change is then only visible at the third position after the decimal point). Another reason is that in some situations the different maintenance activities might cancel each other.

4.5.4.4 Influence of Size of Change As can be derived from the deterioration measures δ and ρ above, adding or deleting primes influences the quality measures. A remaining question is whether the type and size of the change correlates with the type and the size of change of the measures. For the following considerations a relative change in size of the specification is defined and then compared to our quality-based measures.

Definition 18 Relative Deviation of CC. *Let Ψ_{n-1} and Ψ_n be two consecutive versions of a formal Z specification Ψ. Further let $min(\Psi)$ and $max(\Psi)$ be the sizes of the smallest resp. largest versions over all versions of Ψ. Then the relative deviation in size ($\Delta(\Psi_{n-1}, \Psi_n)$) with $n > 1$ is calculated as 1 minus the ratio between the change in size of Ψ_{n-1} to Ψ_n and the maximal difference in size. It is defined as follows:*

$$\Delta(\Psi_{n-1}, \Psi_n) = \begin{cases} 1 + \frac{|\Psi_n| - |\Psi_{n-1}|}{max(\Psi) - min(\Psi)} & \textit{when } max(\Psi) > min(\Psi) \\ 1 & \textit{otherwise} \end{cases}$$

The value of the relative deviation of CC is greater than 1 when primes are added to the specification, and it is less than 1 when primes are removed.

As can be seen in Figures 4.14 and 4.15, a change in size does not always influence our quality measures to the same extent. From the number of additions/modifications/ deletions it is impossible to predict the effect on the measures. Up to revision 1.127, primes are just modified and added to the specification, but the values for *Coverage*, *Overlap*, and *Coupling* sometimes increase and sometimes decrease. Also the amplitude of the changes is no indicator. Around revisions 1.006, 1.046, 1.091 and 1.127 considerable schema modifications took place, however, the biggest changes in *Overlap*, *Coverage* and *Coupling* happened at revisions 1.020, 1.039 and 1.046. On the other hand, at revisions 1.090 and 1.127 the values did not change dramatically.

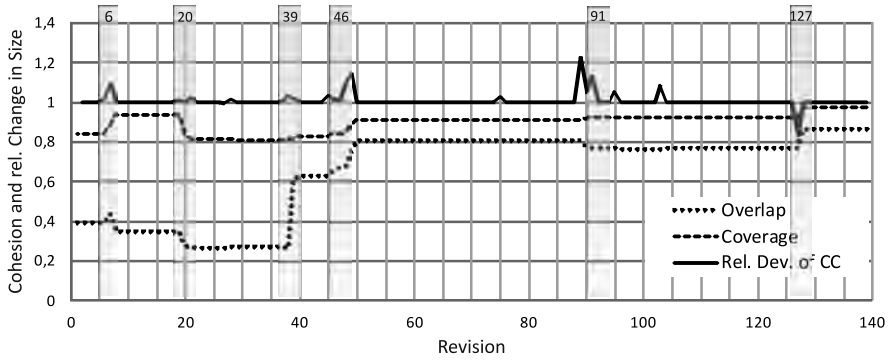


Figure 4.14 Influence in the change of number of primes on the values of cohesion.

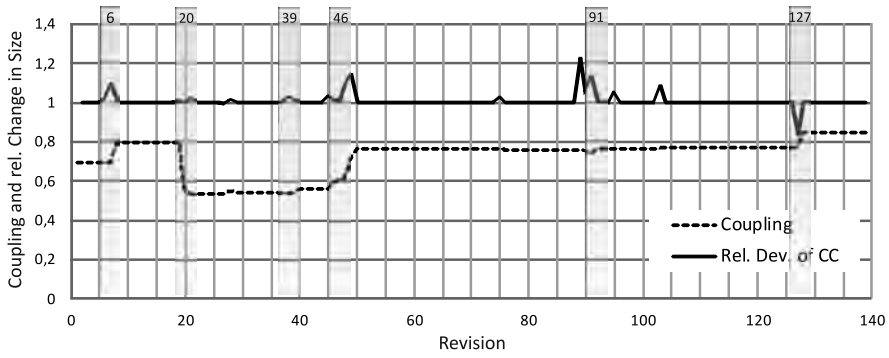


Figure 4.15 Influence in the change of number of primes on the value of coupling.

Measure	$CC(\psi_i)$	$v_l'(\psi_i)$	$v_u'(\psi_i)$	$DU(\psi_i)$	$Cov(\psi_i)$	$O(\psi_i)$	$\chi(\psi_i)$
Mean	52.271	8.623	1,036.341	93.760	0.770	0.640	0.249
σ	80.080	14.642	3,014.316	228.961	0.245	0.396	0.262
95% *CI*	3.565	0.652	134.202	10.194	0.011	0.018	0.012
Range from	48.706	7.971	902.139	83.566	0.759	0.623	0.238
Range to	55.837	9.275	1,170.543	103.954	0.781	0.658	0.261

Table 4.11 Metrics distribution, standard deviation σ and 95% confidence interval *CI* for all 1938 schemas ψ_i.

4.5.5 Baseline

After the head-to-head comparison of the measures and the longitudinal analysis we may take a closer look at baseline metrics. The idea of a baseline is to support separating those schemas of a specification with divergent measures from those that are assumed to be "normal". Table 4.11 summarizes the statistics for all 1938 schemas from our sample collection of specifications. The mean (average) of the values represents the values that can be expected for a "normal" Z specification. The dispersion of the data is illustrated by the values of standard deviation. For example, *overlap* has a standard deviation of 0.396 which means that most of the values fall within a range of 0.792 around the mean value. It also indicates that the measure is quite sensitive as its range is quite broad. On the other hand, *Coverage* has a standard deviation of 0.245, indicating a slightly less sensitive measure (as the range is smaller).

Table 4.11 also provides the values for the 95% confidence interval for the measures. When we can assume that the schemas studied so far represent a sufficiently random sample of the population, then the values demonstrate that the means are quite good representatives. For example, it can be stated that, with a 95% confidence, the average value of *Coverage* falls between 0.759 and 0.781. The other metrics show quite similarly small confidence intervals.

For the sake of completeness the values for size and structural complexity are also provided in Table 4.11. It shows an overview of the distribution of the values, but the mean has to be taken with care as the values for standard deviation are very high. In fact, the specifications differ a lot concerning the number of control and data dependencies.

The baseline (and the measures) can now be used to assess the further development of Z schemas. The approach itself is scalable and can easily be integrated into the development process. All the environment requires as input is a syntactically correct specification, and then it calculates and exports the measures into a CVS file for further examination. The time needed for the calculation depends on the number of dependencies in the $ASRN$ that are to be identified and the number of slices that are to be considered. In our case, the sample specifications have been assessed on a standard laptop (2.67 GHz i7 CPU, 4GB RAM), and the time varied from less than a second ($Birthday\ Book$ specification with 8 slices, $v'_l = 4$, $v'_u = 10$, $DU = 4$), up to 5 minutes ($FlowxSystem$ specification with 2.148 slices, $v'_l = 214$, $v'_u = 45.303$, $DU = 3.476$).

Let us now clarify how the approach can be used. As a first example let us take a look at the $WSDL$ specification. At revision 1.005 the schema $Components$ ($CC = 37$) has an overlap value of 0.443 which is below the mean, and a value of *Coupling* of 0.944. The schema handles all components like $bindingComps$ or $serviceComps$. The $Components$ schema has a lot of different things to deal with, thus the intersection slice contains only a small number of predicates. As the schema is used everywhere, the relationship to other schemas is high. This schema (and with it the handling of the components) has been refactored in the next revision: it has been split into three schemas ($BindingComponents$, $ServiceComponents$, and $OtherComponents$). While *Coupling* decreases to 0.7, the value of *Coverage* increases to 0.549 (for the three schemas).

As another example we can take a look at is the $Tokeneer$ specification. There, the state schema $IDStation$ has a *Coupling* value of 0.431 which is high (also compared to the average value of 0.224 for this specification). The reason is that this schema includes 14 other states. In fact, whenever this schema is mapped to code, it could be assumed that the resulting section in the program code will also be quite complex to handle.

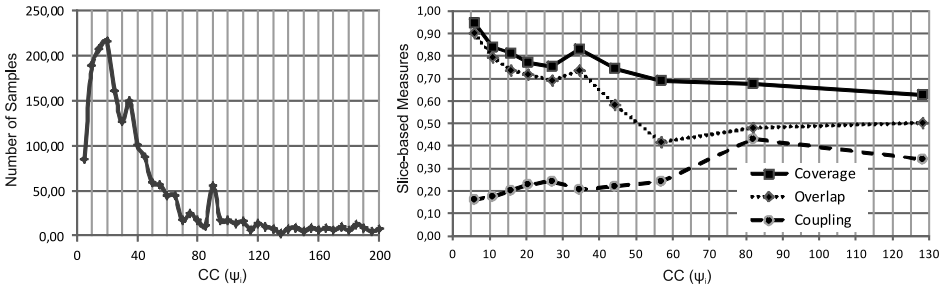


Figure 4.16 (Left side) About 90% of all schemas (1764 of 1938) have a conceptual complexity between 3 and 130. (Right side) Values of slice-based measures for specifications of raising sizes, up to specifications of size CC=130.

Though it is not part of this study, it can be assumed that the baseline values, together with the actual values, can be used for a first orientation as argued above. And yet another property is revealed when looking at the distribution of the means of different specifications: simple (textbook) specifications are typically very compact, do consist of smaller schemas. The values for cohesion are quite high and *Coupling* is usually low. On the other hand, larger specifications are not so dense and they tend to result in lower values for cohesion (and slightly higher values for *Coupling*).

Figure 4.16 makes these tendencies explicit. It visualizes the sizes of the schemas and the baseline. The diagram on the left side shows that about 90% of all schemas have a conceptual complexity between 3 and 130. This is also the reason why the diagram to the right (in the absence of enough specification schemas) is limited to an upper bound of $CC = 130$. Within this range it can be observed that the values for cohesion decrease with increasing sizes of the schemas, whereas the value of *Coupling* increases. *Coupling* will be around an average level of 0.30, the values of cohesion level off at around 0.60. The mean values in Table 4.11 are still of interest, but the variation suggests to adjust them a little depending on the sizes of the specifications at hand.

4.6 Validity

With the results of the empirical study the question of validity arises. The study investigates the relation between (and within) size, structure and semantics-based measures as defined in Sections 4.4.2 and 4.4.3. It does not claim to consider the notion of coupling and cohesion as experienced by developers. In that case human beings would have to be involved, and the notion of validity would have been an issue to be addressed from a different vantage point. As there are no human factors to be regarded, only the following issues have to be considered:

- the selection validity which refers to the degree to which the findings can be generalized,
- the internal validity which is the degree of causal inferences in the empirical study.

Concerning the selection validity, the 35 specifications have been chosen with care as they are from different fields and written by differently experienced authors. The 20.193

lines of specification text of all 1938 Z schemas contain student solutions (55 schemas, 2.8%), textbook exercises (197 schemas, 10.2%) as well as "real world" specifications (1686 schemas, 87.0%). The "real world" examples are from practitioners as well as from academic authors and the diversity of the specifications makes it more likely that the results are valid for a wider class of formal specifications. In addition to that, nearly all of the specifications used in this study are publicly available. This makes it easier, if necessary, to refer to them and to verify the results later on.

It is important to note that some of the older specifications used in this study had to be modified a bit in order to be accepted by the CZT parser. The modifications were done by hand and only dealt with syntactical improvements (hard-spaces) so that the parser was able to read the text without syntax errors. Eventually, the "$\widehat{=}$" symbol had to be replaced by the "$==$" sign to satisfy the Z grammar of the CZT parser. These modifications were carried out with care. In order to rule out the possibility of coincidental changes of line breaks or identifier names, all of the files were compared again afterwards, using a professional file-compare software.

Considering internal validity, single group and multiple group threads, as well as social threads cannot arise. The only thread that might influence the outcome of the study is the software system used to generate and calculate the measures. The software components involved are the CZT parser, the slicing environment ViZ, $Matlab\ R2007b$, and $SPSS$ 14.0. $Matlab$ and $SPSS$ are numerical computer environments used for the statistical analysis. Both tools have been used alternately to verify the results of the analysis. It is very unlikely that the data from both environments is erroneous. It is more likely that problems arise due to the parser or slicer.

The CZT parser is being developed as a $sourceforge$ project since the year 2003 and is the basis of a lot of extensions[3] (for example the Z animations tool $JaZa$ or the $Eclipse$ plug-in). Irrespective of these various enhancements the parser is stable. What remains is the slicing environment that might not have worked as expected. ViZ has also been developed in the year 2003 and is also a fundamental part of a couple of extensions. During summer 2011 it has been upgraded to make use of the CZT $1.5.0$ version and it is now also able to deal with the dot-notation correctly. With this new version the whole syntax of Z can be parsed. The results of the slicer have been validated systematically during development, on the one hand by comparing the outcome of the slicing environment to documented results to be found in literature, and on the other hand by structured paper-and-pencil tests. Even if this does not totally impede the threat, it reduces at least the chance that implementation faults influence the results.

4.7 Related Work

The basis for the approach presented is the reconstruction of control and data dependencies within Z specifications. The papers influencing this study have already been mentioned in Section 4.3. Basically, the general idea goes back to the work of Weiser [50, 51], who showed how to compute the set of program statements that affect program values at some point of interest (called the slicing criterion), and to the work of Meyers and Binkley [35], who did the same analysis as in this work, but for a large collection of "traditional" programs.

[3]For more details on the CZT $sourceforge$ project see http://czt.sourceforge.net. Page last visited: Nov. 2010.

Concerning formal specifications, the first approach of "specification slicing" goes back to Oda and Araki [38]). Their idea has been extended and re-defined by Chang and Richardson, Bollin, and Wu [12, 3, 54]. An extension of the slicing approach to Object-Z specifications has been introduced by Brückner and Wehrheim [11]. They also make use of a "Program" dependence graph that is built from control and data dependencies.

Other specification comprehension techniques that make use of intra-schema relations are those of the transformations to other types of notations [31, 28, 18] or the application of diverse concept location techniques [5].

Papers that deal with the measurement of specifications are rare. One of the first studies that took a closer look at specification measures was conducted by Samson, Nevill and Dugard in 1987. It used examples to show that there is a correlation between specification metrics and metrics of the deferred implementation [45]. The authors demonstrated how, by counting the number of equations, an estimate of effort is possible at a much earlier stage in the development process.

The often cited CDIS project [41] demonstrated that formal design yields a highly reliable code. The authors showed that formal methods are very effective in acting as a catalyst for testing. However, the main measures in the CDIS project were the lines of specification text and the time needed to write the specifications.

In 1996, Clarke and Wing [14] presented a survey of the use of formal specifications and existing tools. More than 120 references are provided, giving the reader a sense of acceptability of formal methods. Unfortunately, the measures they address are only related to the quality of the final software systems, the development costs and time, and the productivity rate in the projects.

In 2002, an empirical study conducted by Sobel and Clarkson [46] showed that the application of formal analysis provides great benefit to implementation with respect to completeness. One outcome of the study was that the group using formal methods passed nearly 100% of the standard set of test cases in comparison to 45.5% passed by the control teams. Again, the measurement focussed on the final products and not on the specifications.

The list of formal specification projects can be continued (a collection can be found at the Formal-Methods-Wiki page of Bowen [8]), but quality and complexity measures of the underlying formal specifications are, apart from size-based measures, not considered in the available documentation.

4.8 Summary

A lack of suitable measures can diminish the benefits of an idea or approach. This article, therefore, takes a closer look at measures of formal Z specifications in order to broaden the field of application and acceptability. The three key contributions of the empirical study are:

- The study provides a head to head comparison of size-/structure- and semantics-based measures. It demonstrates which of these measures offer unique views of the specification and one result is that slice-based measures are especially good and sensitive descriptors for a specification at hand.
- The study takes a closer look at the evolution of a specification. In a longitudinal study of the Web Service Definition Language it shows that slice-based measures can also be used for assessing deterioration effects of formal specifications.

- Last but not least, the study takes a closer look at the different measures and gives a summary of their means and standard deviations. As the values vary for specifications of different sizes and types, it also provides a baseline for the measures.

The objective of this contribution was not to invent new measures, but to examine whether or not the idea of using dependencies and slices to measure inter- and intra-connectivity also holds for state-based formal specifications. To some extent this question can be answered positively as parts of the measures describe unique properties. The longitudinal study of the *WSDL* specification indicates the usefulness of the measures, but it is to be seen to which extent these results will be supported by other cases having an extended life-time of evolving specifications. Likewise, adaptations of the quality measures are still issues for further research, and a more comprehensive study on this issue is certainly warranted.

Acknowledgement

I am very grateful for the remarks of Roland Mittermeir and John Brown, who's constructive remarks motivated me to elaborate further on this topic.

REFERENCES

[1] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996.

[2] David Basin, Hironobu Kuruma, Kazuo Takaragi, and Burkhart Wolff. Specifying and Verifying the Hysteresis Signature System with HOL-Z. Technical Report 471, ETH Zr̈ich, 2004.

[3] Andreas Bollin. *Specification Comprehension – Reducing the Complexity of Specifications*. PhD thesis, Universität Klagenfurt, April 2004.

[4] Andreas Bollin. Maintaining Formal Specifications. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary*, pages 442–453, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[5] Andreas Bollin. Concept Location in Formal Specifications. *Journal of Software Maintenance and Evolution – Research and Practice*, 20(2):77–105, 2008.

[6] Andreas Bollin. Slice-based Formal Specifiation Measures – Mapping Coupling and Cohesion Measures to Formal Z. In Cèsar Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium*, NASA/CP-2010-216215, pages 24–34. NASA, Langley Research Center, April 2010.

[7] Jonathan Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press (ITCP), London, UK, 1996.

[8] Jonathan P. Bowen. Formal Methods Wiki. http://formalmethods.wikia.com. Page last visited Oct 9th, 2010, February 2010.

[9] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, July 1995.

[10] Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. HOL-Z 2.0: A Proof Environment for Z-Specifications. *Journal of Universal Computer Science*, 9(2):152–172, 2003.

[11] Ingo Brckner and Heike Wehrheim. Slicing object-z specifications for verification. In *In ZB 2005, Volume 3455 of LNCS*, pages 414–433. Springer-Verlag, 2005.

[12] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[13] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0. http://www.w3.org/TR/wsdl20, 2007.

[14] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions, CMU Computer Science Technical Report CMU-CS-96-178. Technical report, Carnegie Mellon University, August 1996.

[15] David Cooper and Janet Barnes. Tokeneer ID Station – EAL5 Demonstrator: Summary Report. S.p1229.81.1, Praxis High Integrity Systems, 2008.

[16] Maximiliano Cristià and Miriam Alvez. A collection of sample specifications for the test template framework fastest. Contact: cristia@cifasis-conicet.gov.ar, CIFASIS, Universidad Nacional de Rosario, Flowgate Security Consulting, Argentina, October 2011.

[17] Antoni Diller. *Z – An Introduction to Formal Methods*. John Wiley and Sons, 1999.

[18] Houda Fekih, Leila Jemni Ben Ayed, and Stephan Merz. Transformation of B specifications into UML class diagrams and state machines. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1840–1844, New York, NY, USA, 2006. ACM.

[19] N.E. Fenton and A.A. Kaposi. An Engineering Theory of Structure and Measurement. In B.A. Kitchenham and B. Littlewood, editors, *Software Metrics. Measurement for Software Control and Assurance*, pages 27–62. Elsevier, London, UK, 1989.

[20] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics*. Thompson Press, 2nd edition, 1989.

[21] Leo Freitas, Mark Utting, Petra Malik, and Tim Miller. SourceForge Project – Community Z Tools. http://sourceforge.net/projects/czt. Page last visited: Oct. 2010, 2010.

[22] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[23] Anthony Hall, David L. Dill, John Rushby, C. Michael Holloway, Ricky W. Butler, and Pamela Zave. Industrial Practice - Impediments to Industrial Use of Formal Methods. *IEEE Computer*, 29(4):22–27, April 1996.

[24] Mark Harman, Margaret Okulawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of coupling. In *Proceedings of the IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution. Boston, Massachusetts*, pages 28–32, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[25] Ian Hayes, Bill Flinn, Roger Gimson, Steve King, Carroll Morgan, Ib Holm Sørensen, and Bernard Sulfrin. *Specification Case Studies*. Prentice Hall International, Hertfordshire, UK, 1992.

[26] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):1–76, 2009.

[27] Mike Hinchey, Michael Jackson, Patrick Cousot, Byron Cook, Jonathan P. Bowen, and Tiziana Margaria. Software engineering and formal methods. *Commun. ACM*, 51(9):54–59, 2008.

[28] Akram Idani and Yves Ledru. Object Oriented Concepts Identification from Formal B Specifications. *Electronic Notes in Theoretical Computer Science*, 133:159–174, May 2005.

[29] Jonathan Jacky, Michael Patrick, and Jonathan Unger. Formal specification of control software for a radiation therapy machine. Technical report, Technical Report 95-12-01, Radiation Oncology Department, University of Washington, Seattle, WA, 1997.

[30] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, Upper Saddle River, NJ 07458, USA, 2^{nd} edition, 1990.

[31] Soon-Kyeong Kim and David Carrington. A Formal Mapping between UML Models and Object-Z Specifications. *Lecture Notes in Computer Science*, 1878:2–21, 2000.

[32] Herbert D. Longworth. Slice Based Program Metrics. Master's thesis, Michigan Technological University, 1985.

[33] Petra Malik. A retrospective on czt. *Software – Practice and Experience*, 41(2):179–188, 2011.

[34] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[35] Timothy M. Meyers and David Binkley. An Empirical Study of Slice-Based Cohesion and Coupling Metrics. *ACM Transactions on Software Engineering and Methodology*, 17(1):2:1–2:27, December 2007.

[36] Tim Miller, Leo Freitas, Petra Malik, and Mark Utting. CZT Support for Z Extensions. In *Proc. 5th International Conference on Integrated Formal Methods (IFM 2005)*, pages 227 – 245. Springer, 2005.

[37] Roland Mittermeir and Andreas Bollin. Demand-Driven Specification Partitioning. In László Böszörmenyi and Peter Schojer, editors, *Modular Programming Languages*, volume 2789 of *Lecture Notes in Computer Science*, pages 241–253. Springer Berlin / Heidelberg, 2003.

[38] Tomohiro Oda and Keijiri Araki. Specification slicing in a formal methods software development. In 17^{th} *Annual International Computer Software and Applications Conference*, IEEE Computer Socienty Press, pages 313–319, November 1993.

[39] Linda M. Ott and Jeffrey J. Thus. The Relationship between Slices and Module Cohesion. In *11th International Conference on Software Engineering*, pages 198–204, Los Alamitos, CA, USA, 1989. IEEE Computer Society.

[40] Linda M. Ott and Jeffrey J. Thuss. Slice Based Metrics for Estimating Cohesion. In *In Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81, Los Alamitos, CA, USA, 1993. IEEE Computer Society.

[41] Shari Lawrence Pfleeger and Les Hatton. Investigating the Influence of Formal Methods. *IEEE Computer*, 30(2):33–43, Feb. 1997.

[42] D. G. Rees. *Essential Statistics*. Chapman & Hall, 4th edition, 2003.

[43] Philip E. Ross. The Exterminators. *IEEE Spectrum*, 42(9):36–41, September 2005.

[44] Arthur Ryman. Concurrent Versioning System Log for WSDL 2.0. http://dev.w3.org/cvs/web/2002/ws/desc/wsdl20/Attic/wsdl20.tex, 2007.

[45] W.B. Samson, D.G. Nevill, and P.I. Dugard. Predictive software metrics based on a formal specification. In *Information and Software Technology*, volume 29 of *5*, pages 242–248, Newton, MA, USA, June 1987. Butterworth-Heinemann.

[46] Ann E. Kelley Sobel and Michael R. Clarkson. Formal Methods Application: An Empirical Tale of Software Development. *IEEE Transaction on Software Engineering*, 28(3):308–320, March 2002.

[47] J.M Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, Upper Saddle River, NJ 07458, USA, 2^{nd} edition, 1992.

[48] Kuo-Chung Tai. A program complexity metric based on data flow information in control graphs. *Proceedings of the 7th International Conference on Software Engineering*, pages 239–248, 1984.

[49] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[50] Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.

[51] Mark Weiser. Program slicing. In *Proceedings of the* 5^{th} *International Conference on Software Engineering*, pages 439–449, Piscataway, NJ, USA, 1982. IEEE Press.

[52] Jim Woodcock and Jim Davis. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, July 1996.

[53] Jim Woodcock and Leo Freitas. *AN ELECTRONIC PURSE (in Z/Eves) – Specification, Refinement, and Proof*. Oxford University Computing Laboratory, Department of Computer Science (HISE Group), Heslington YO10 5DD, UK, 2006.

[54] Fangjun Wu and Tong Yi. Slicing Z Specifications. *ACM SIGPLAN Notices*, 39(8):39–48, 2004.

CHAPTER 5

PREDICTIVE SOFTWARE MEASURES BASED ON Z SPECIFICATIONS – A CASE STUDY

A. Bollin, A. Tabareh

Submitted to the 2^{nd} Workshop on Formal Methods in the Development of Software, co-located with the 18^{th} International Symposium on Formal Methods, CNAM Paris, France, 8 pages, August 2012.

5.1 Introduction

Recent studies in the areas of software metrics and project management have stimulated a lot of ideas of how development effort can be estimated and which metrics are of relevance [15, 9, 19]. Basically, they all suggest that the collection of data and the estimation process should be performed as early and as objectively as possible – so why not taking a closer look at properties of formal specifications?

To the best of our knowledge, the only publicly available case study that took a closer look at correlations between specifications and implementations was conducted by Samson, Nevill and Dugard in 1987 [17]. The authors used Modula-2 modules and a HOPE specification to show that there is a correlation between the number of equations in HOPE and the number of lines of source code and cyclomatic complexity in the Modula-2 modules. However, the authors admit that the study is relatively small-scale as their data is based on only 9 experimental subjects.

The objective of this paper is now to shed some more light onto the question whether specifications' properties can help predicting attributes of derived implementations or not. For this, the following strategy is pursued: firstly, based on a set of well-known measures, it tries to find out whether some of the measures are correlated or not. Secondly, it suggests

a prediction model for some of the measures. A case study, based on the specification and implementation of the $Tokeneer$ system [5] forms the basis for these considerations. It takes the Z specification of the system and its implementation in ADA as the point of departure and identifies those parts of the code that unambiguously implement specific parts of the specification. After that, it calculates size, structure and quality related measures for both of the documents. Finally, it looks for correlations between the measures, and, based on the findings, it calculates a prediction model for several ADA-based size- and complexity-related measures.

This paper is structured as follows: Section 5.2 briefly introduces the code and specification measures that are used in the study. Section 5.3 presents the setting of the study, the experimental subject and the statistical tests used. Next, Section 5.4 presents and discusses the results of the correlation tests, and Section 5.5 presents the prediction model. Section 5.6 discusses possible threats to validity, and, finally, Section 5.7 summarizes the findings and discusses possible steps to be done next.

5.2 Measures

This section introduces the measures used for assessing the Z specification and its implementation in ADA. Please note that, due to limitations of space, only a brief overview of the measures is provided.

5.2.1 Code-based Measures

The implementation language of the $Tokeneer$ specification [4] is ADA. In his master thesis, Tabareh [20] took a look at currently available environments that are able to generate practical measures from ADA code. He suggests to apply the $Understand$ tool and uses the following measures (where M denotes either an ADA function or a procedure)[1] for a preliminary study comparing ADA and Z-based measures:

- CountLine $CL(M)$. It counts the number of physical lines.
- CountLineCode $CLC(M)$. It counts the number of lines that contain source code.
- CountLineExecutable $CLCE(M)$. It counts the number of lines containing executable ADA code.
- CountLineCodeDecl $CLCD(M)$. It count the number of lines containing declarative ADA code.
- Knots Count $KNOTS(M)$: It is a measure for the structuredness of a module and counts overlapping jumps in the program flow graph.
- Cyclomatic Complexity $CYC(M)$. It measures the maximum number of linearly independent paths through a program and is extracted by counting the minimum set of paths which can be used to construct all other paths through the graph.

In order to focus even more on structural properties of the code, this study additionally makes use of Shepperd and Ince's Information flow count [18]. The general idea is that the complexity of a module is related to the number of flows or channels of information between the module and its environment. For this, the $Understand$ tool can be used

[1]The tool and a description of the measures can be found at the Understand homepage at www.scitools.com. Page last visited: May 2012.

to generate the call-graph, and the flow of data and control then forms the basis for the calculation of the Sheppard Information Flow SI of a module M:

- Fan-in ($FIN(M)$): It comprises the number of data-flows terminating at a component M.
- Fan-out ($FOUT(M)$): It comprises the number of data-flows originating at a component M.
- Information Flow($SI(M)$): It comprises the number of information flows related to a component M and is calculated via $(FIN(M) * FOUT(M))^2$.

5.2.2 Specification Measures

Most of the complexity measures for formal specifications focus on size. The reasons are that size-based measures (like lines of specification text) are easy to calculate and yield a single number that is easy to interpret. This is not so much the case for structure- and quality-related measures. Their calculation is usually based on the notion of control and data dependencies, concepts that are not necessarily dominant principles of a specification language. However, several authors [13, 3, 12] demonstrated that a reconstruction of these dependencies is possible.

Recently, Bollin showed that coupling and cohesion based measures can reasonably be mapped to formal Z specifications [2]. The basis for the calculation of all the measures is a graph that contains vertices (called $primes$) for all predicates and declarations of the specification, and arcs representing (reconstructed) control and data dependencies [1]. With such a graph as a basis, the following measures (defined for schemas ψ that are part of a specifications Ψ) are used in the remainder of this work:

- Conceptual Complexity $CC(\psi)$: The conceptual complexity equals the total number of prime vertices in the graph (representing a schema ψ).
- Logical Complexity $v'(\psi) = (l, u)$: The lower bound value l of the measure is 1 plus the number of primes that are terminal vertices of control dependency arcs. It can be compared to counting the number of decision statements in programs. The upper bound value u equals 1 plus the total number of control dependencies. It reflects the total amount of dependencies to be considered.
- Definition-Use Count: $DU(\psi)$: This measure equals the total number of data dependencies.
- Use Count $USE(\psi)$: The use count equals the number of identifiers used in the schema ψ.
- Definition Count $DEF(\psi)$: The definition count equals the number of identifiers referring to an after state in the schema ψ of the specification.
- And-Count $AND(\psi)$: This measure equals the number of AND-combined predicates in ψ.
- Or-Count $OR(\psi)$: This measure equals the number of OR-combined predicates in ψ.

Semantics-based measures can be calculated by generating slices. The idea goes back to the work of Weiser [21, 22] who introduced five slice-based measures for cohesion: Tightness, Coverage, Overlap, Parallelism and Clustering. Ott and Thuss [14] partly formalized these measures. Coupling, on the other hand, was originally defined as the number of local information flow entering (fan-in) and leaving (fan-out) a procedure [8]. Harman et. al

[7] demonstrate that it can also be calculated via slicing. Mapping and evaluating their approaches to Z leads to the following set of quality-based specification metrics [2]:

- *Coverage* $Cov(\psi)$: It measures the compactness of a schema by comparing the length of all possible slices to the length of the specification schema ψ.
- *Overlap* $O(\psi)$: It measures the conciseness of a schema ψ by counting those statements that are common to all of the possible slices and relates the number to the size of all slices.
- *Schema Coupling* $\chi(\Psi, \psi_i)$: It is the weighted measure of the information flow between a given schema ψ_i and all other schemas in Ψ.

5.3 The Study

The study is split into two parts and it aims at answering the following two questions: (a) what type of correlations exists between specification-based and code-based measures, and (b) is it possible to predict code-based measures from specification-based measures?

The Tokeneer system [4] is one of the rare, industrial-size and publicly available, formal Z specifications that comes along with a fully derived implementation. It has been developed by Praxis and the NSA and provides a specification for an identification station consisting of a fingerprint reader, a display and a card reader. The code, written in ADA, consists of $11,807$ lines of executable ADA code ($34,769$ lines including comments). The exceptional feature of the ADA files is that they contain so-called "trace unit" comments which are direct links to the corresponding sections in the formal design document, thus linking specification text (schemas) and implementation code pairs (procedures and functions) unambiguously together. The Z specification consists of $11,356$ lines of text, including $4,808$ lines of specification text. The specification itself contains $3,295$ declarations and predicates, it contains $132,088$ control dependencies and $6,145$ data-dependencies.

The subjects for this study are set of pairs of code modules (procedures and functions) and their related Z specification (schemas). However, the mapping is not always one-to-one, and it is also not total. There are a couple of trace-units that do not have a corresponding part in the implementation, and there are also links to trace-units that are non-existent. Thus, as a first step in the preparation phase of this study, a small script was written for matching the references and units automatically, sorting out spelling errors and dangling links. Then, the result of the mapping has been verified and cross-checked by hand. This process yielded 70 units with a traceable transformation of Z code to ADA code.

The first part of the study deals with the question of relatedness between specification-code pair measures. As we do not know whether the measures are normally distributed, three different statistical tests are used to assess the data: the Pearson's Correlation Coefficient, the Spearman's Rank Correlation Coefficient, and Kendall's Tau Correlation Coefficient. The Pearson's correlation coefficient (R_P) measures the degree of association between the variables, assuming normal distribution of the values [16, p. 212]. Though this test might not necessarily fail when the data is not normally distributed, the Pearson's test only looks for a linear correlation. It might indicate no correlation even if the data is correlated in a non-linear manner. As the data might not be normally distributed, the Spearman's rank correlation coefficient (R_S) has been chosen [16, p. 219]. It is a non-parametric test of correlation and assesses how well a monotonic function describes the association

between the variables. This is done by ranking the sample data separately for each variable. Finally, the Kendall's robust correlation coefficient (R_K) is used as an alternative to the Spearman's test [6, p. 200]. It is also non-parametric and investigates the relationship among pairs of data. However, it ranks the data relatively and is able to identify partial correlations.

When a value of $\mid R \mid \in [0.8,\ 1.0]$ then it is interpreted to indicate a *strong association*. When $\mid R \mid \in [0.5,\ 0.8)$ it is interpreted to indicate a *moderate association*. When $\mid R \mid \in [0.0,\ 0.5)$ it is interpreted to indicate a *weak association* (values rounded to the third decimal place).

5.4 Correlation Tests

After data preparation, the study looked for linear or at least partial correlations between the sets of measures. At first, classical size-based measures are considered, and Table 5.2 summarizes the results. The p-values for testing the hypothesis of no correlation against the alternative that there is a nonzero correlation are less than 0.05 for all tests. The table also shows that there is a moderate to strong relation between $CC(\psi)$ and the measures of $CL(M)$, $KNOTS(M)$ and $FOUT(M)$. The correlation values of the tests are quite similar, but there are a couple of exceptions. Compared to the Pearson test, the Spearman's rank test shows a higher correlation between most of the size-based measures and the Count Line Declarative $CLCD(M)$ measure, indicating that there might be a non-linear correlation between them. However, the Kendall's test shows weak correlation for most of the measures again. A similar situation can be observed for the measure of $FIN(M)$. Here, the Spearman test shows a slightly higher correlation than the other two tests, but it still falls into the weak association class. Interesting are the differences between the tests for the $SI(M)$ measure. The correlation to the size-based measures is not strong, but $SI(M)$ is calculated by also using the square of $FOUT(M)$, and this non-linear tendency can be seen in the slightly higher values of the Spearman tests. And yet another issue can be observed: cyclomatic complexity is (although only moderately) influenced by the number of logical OR connections in the specification. As cyclomatic complexity is related to the number of paths through the program, this observation seems also to be consistent to the use of or-combined predicates in a Z specification.

In a second step, structure-based measures have been looked at. Table 5.3 summarizes the results. Again, the p-values are less than 0.05 for all tests. The correlations are not as strong as with the pure size-based measures – with one exception: the structure-based measures seem to strongly influence the $FOUT(M)$ count. The other structure-based measures moderately to strongly influence the complexity measures $CYC(M)$ and $KNOTS(M)$. This seems to be inherent, as these measures are counting dependencies within and between the schemas. The correlation to the other ADA-related measures in also moderate to strong. Only the measures of $CLCD(M)$, $FIN(M)$ and $SI(M)$ do have weak correlations.

In the case of semantics-based measures the picture has to be looked at in a more differentiated way (see Table 5.4). At first, most of the results of the tests concerning *Coverage* are statistically not significant (higher p values are shown in bold numerals). The tests indicate no correlation between the ADA-based measures and *Coverage*, but the chance is high that this is wrong. In this situation scatter plots have been used to gain a better understanding of the results, but the plots confirmed the results of no correlation at all. The

Results of the backward elimination procedure (values $P \leq 0.4$)						
Paramter		*CL(M)*	*CLE(M)*	*CYC(M)*	*KNOTS(M)*	*FOUT(M)*
Adjusted R-Square		0.720	0.680	0.620	0.760	0.840
Significance F		5E-18	2E-16	5E-14	7E-20	1.5E-25
$P - Value$	$CC(\psi)$	0.001	4E-4	0.003	1E-6	3E-11
	$v_l'(\psi)$	——	——	——	——	0.270
	$v_u'(\psi)$	——	0.005	——	0.004	0.000
	$DU(\psi)$	——	——	——	——	——
	$O(\psi)$	——	——	——	——	——
	$Cov(\psi)$	——	——	0.280	——	——
	$\chi(\psi)$	——	——	——	——	——
	$AND(\psi)$	7E-5	——	——	0.030	——
	$OR(\psi)$	5E-4	0.001	4E-4	——	6E-5
	$DEF(\psi)$	——	0.070	——	0.020	1E-5
	$USE(\psi)$	0.034	——	——	0.320	——

Table 5.1 Adjusted R-Square, Significance F and P-Values after applying the backward elimination procedure for maximum model identification. P-Values higher than 0.4 are represented by dashes.

other tests indicate weak to moderate relations for *Overlap* and *Coupling*, but another point is interesting. *Overlap* and *Coupling* have different leading signs. This might partially be explained by the fact that overlap is an indicator for crispness. It is high when the schema is not strongly related to other parts of the specification. And coupling is higher when there are more relations to other specification schemas. An increase in the value of one measure leads to a decrease of the other measure.

To summarize, there is only weak to moderate relation between the Z-based measures and $CLCD(M)$, $FIN(M)$, and $SI(M)$. But, though not exclusively, there is some moderate to strong correlation between the Z-based and the other implementation-based measures. When just focusing, for example, on $CL(M)$, $CYC(M)$, $KNOTS(M)$, and $FOUT(M)$ and taking moderate to strong correlations into account, then the following can be observed: Firstly, they are all influenced by structure-based measures. And, secondly, especially $CL(M)$, $KNOT(M)$ and $FOUT(M)$ do have strong correlations to the Z measures. The next section now takes the Z measures and uses them to provide regression formulas for the most suitable ADA-based measures.

5.5 Prediction Models

According to a rule of thumb in regression [10, p.3], the appropriate number of independent variables for a prediction is not more than one fifth of the sample size. Thus, the eleven Z measures presented in Section 5.2 can be considered to be sufficient and they are all selected to form the maximum model for 70 observations in this study. Among several systematic methods for restricting the maximum model, a backward elimination procedure

[10, p.8] with a threshold of 0.4 for the P-values is selected. This means that a maximum regression model with all eleven independent variables is built. Then all the variables with a P-value of more than 0.4 are eliminated. Then, again, another regression model with the reduced number of variables is built, iterating until there is no variable with a P-value higher than 0.4.

Table 5.1 summarizes the final result of this procedure for the five remaining code metrics (as measures with a P-Value higher than 0.4 have been eliminated). The table, for example, shows that for the calculation of the cyclomatic complexity $CYC(M)$ of an ADA module, $CC(\psi)$, $Cov(\psi)$ and $OR(\psi)$ are best for being used in the regression formula. The level of confidence can be explained by the values of *Significance-F*. If the level of acceptable confidence should be 95% and higher, then all the code metrics with F-values of less than 0.05 can be considered predictable using the metrics in Z. All the values for F in Table 5.1 show that there is a high reliability on the results of the regressions. The value of *Adjusted R-Square* can be interpreted as an indicator for the precision level of the prediction. In our case the values are between 0.620 and 0.840, indicating that the regression models are relatively precise for $FOUT(M)$, $KNOTS(M)$ and $CL(M)$ and even more precise for $CLE(M)$ and $CYC(M)$. With these values at hand it makes sense to predict code metrics, and the resulting formulas are as follows:

$$CL(M) = 3.099\,CC(\psi) - 1.237\,USE(\psi) + 2.557\,AND(\psi) - 41.735\,OR(\psi) - 9.873$$
$$CLCE(M) = 0.516\,CC(\psi) - 0.003\,v_u'(\psi) - 0.477\,DEF(\psi) + 5.458\,OR(\psi) + 5.819$$
$$CYC(M) = 0.015\,CC(\psi) + 4.349\,Cov(\psi) - 2.107\,O(\psi) + 1.082\,OR(\psi) + 1.666$$
$$KNOTS(M) =$$
$$0.121\,CC(\psi) - 0.001\,v_u'(\psi) - 0.017\,USE(\psi) - 0.092\,DEF(\psi) + 0.027\,AND(\psi) - 0.882$$
$$FOUT(M) = 0.198\,CC(\psi) - 0.107\,v_l'(\psi) - 0.001\,v_u'(\psi) - 0.211\,DEF(\psi) + 1.220\,OR(\psi) + 0.344$$

5.6 Threats to Validity

With the results of the study the question of validity arises. Considering internal validity, single group and multiple group threats, as well as social threats cannot arise. The only threat that might have an impact on the outcome of the study is the software used to generate and calculate the measures. The software components involved are the CZT parser [11], the slicing environment ViZ [1], $Matlab\ R2007b$, $Microsoft\ Excel$ 2010 and $SPSS$ 14:0. $Excel$ is a standard spreadsheet application. $Matlab$ and $SPSS$ are numerical computer environments used for the statistical analysis. Both tools have been used alternately to verify the results of the analysis. It is very unlikely that the data from both environments is erroneous. The CZT parser is being developed as a $SourceForge$ project since 2003 and it is available in a stable release. The slicing environment ViZ has been developed in the year 2003 and it is also part of a couple of extensions which led to systematic validations during development.

Concerning the selection validity, the publicly available schemas and ADA procedures and functions have been chosen with care, following the links provided by the developers. It is important to note that the specification used in this study had to be modified a bit in order to be accepted by the CZT parser. This meant to introduce some hard spaces and, eventually, also to replace the "$\widehat{=}$" symbol by the "$==$" sign. In order to rule out the possibility of coincidental changes of line breaks or identifier names, both files were again compared afterwards, using a professional file-compare software.

5.7 Conclusion

In this study, consisting of 70 experimental subjects, the feasibility of confidently predicting software measures based on formal specifications has been demonstrated. The correlations found between the different size-, structure-, and semantics-based measures and the implementation metrics promise of being able to predict size and complexity attributes as well as enables to estimate likely costs and efforts.

The study describes only the first link in the chain of associations between the documents created during software development, but it confirms the observations of Samson et.al. [17] who conducted a similar study (with 9 experimental subjects) several years ago. Specification-based measures are not difficult to calculate, thus they can and also *should* be collected at the beginning of a project. The results of the study indicate that it pays off.

REFERENCES

[1] Andreas Bollin. Concept location in formal specifications. *Journal of Software Maintenance and Evolution: Research and Practice*, Manuscript submitted Jan. 2007, 2007.

[2] Andreas Bollin. Slice-based Formal Specifiation Measures – Mapping Coupling and Cohesion Measures to Formal Z. In Cèsar Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium*, NASA/CP-2010-216215, pages 24–34. NASA, Langley Research Center, April 2010.

[3] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[4] Rod Chapman. *The Tokeneer ID Station – Overview and Readers Guide. S.P1229.81.8. Issue: 1.4*. Praxis High Integrity Systems, May 2009.

[5] David Cooper and Janet Barnes. Tokeneer ID Station – EAL5 Demonstrator: Summary Report. S.p1229.81.1, Praxis High Integrity Systems, 2008.

[6] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics*. Thompson Press, 2nd edition, 1989.

[7] Mark Harman, Margaret Okulawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of coupling. In *Proceedings of the ICSE workshop on Process Modelling and Empirical Studies of Software Evolution. Boston, Massachusetts*, pages 28–32, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[8] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

[9] Magne Jørgensen. A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, 70(1–2):37–60, 2004.

[10] Pia Veldt Larsen. *ST111: Regression and analysis of variance – Module 8: Selecting regression models*. Manual from statmaster.sdu.dk/courses/st111/module08, Syddansk University, Department of Statistics, 2008.

[11] Petra Malik. A retrospective on czt. *Software – Practice and Experience*, 41(2):179–188, 2011.

[12] Roland T. Mittermeir and Andreas Bollin. Demand-driven specification partitioning. In *Proceedings of the 5th Joint Modular Languages Conference, JMLC'03*, pages 241–253, Ausgust 2003.

[13] Tomohiro Oda and Keijiri Araki. Specification slicing in a formal methods software development. In *Seventeenth Annual International Computer Software and Applications Conference*, IEEE Computer Socienty Press, pages 313–319, November 1993.

[14] Linda M. Ott and Jeffrey J. Thus. The Relationship between Slices and Module Cohesion. In *11th International Conference on Software Engineering*, pages 198–204, Los Alamitos, CA, USA, 1989. IEEE Computer Society.

[15] Lawrence H. Putnam and Ware Myers. *Five Core Metrics: The Intelligence Behind Successful Software Management*. Dorset House, 2003.

[16] D. G. Rees. *Essential Statistics*. Chapman & Hall, 4th edition, 2003.

[17] W.B. Samson, D.G. Nevill, and P.I. Dugard. Predictive software metrics based on a formal specification. In *Information and Software Technology*, volume 29 of *5*, pages 242–248, June 1987.

[18] Martin J. Shepperd and Darrel C. Ince. The use of metrics in the early detection of design errors. In *Proceedings of the European Software Engineering Conference 90*, pages 67–85, 1990.

[19] Harry M. Sneed, Richard Seidl, and Manfred Baumgartner. *Software in Zahlen*. Carl Hanser Verlag, 2010.

[20] Abdollah Tabareh. Predictive Software Measures Based on Formal Z Specifications. Master's thesis, University of Gothenburg - Department of Computer Science and Engineering, September 2011.

[21] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, 1979.

[22] Mark Weiser. Program slicing. In *Proceedings of the* 5^{th} *International Conference on Software Engineering*, pages 439–449, Piscataway, NJ, USA, 1982. IEEE Press.

Pearson Correlation R_P, $n = 70$, $p \leq 0.033$									
Measure	*CL(M)*	*CLC(M)*	*CLCD(M)*	*CLCE(M)*	*CYC(M)*	*KNOTS(M)*	*FIN(M)*	*FOUT(M)*	*SI(M)*
$CC(\psi)$	0.806	0.681	0.343	0.797	0.749	0.842	0.258	0.866	0.255
$AND(\psi)$	0.538	0.507	0.369	0.519	0.586	0.477	0.366	0.482	0.384
$OR(\psi)$	0.450	0.616	0.435	0.640	0.697	0.490	0.456	0.628	0.481
Spearman's Rank Correlation R_S, $n = 70$, $p \leq 0.016$									
$CC(\psi)$	0.784	0.742	0.653	0.797	0.770	0.783	0.418	0.849	0.615
$AND(\psi)$	0.398	0.428	0.406	0.457	0.454	0.425	0.286	0.452	0.343
$OR(\psi)$	0.697	0.731	0.699	0.760	0.748	0.704	0.497	0.774	0.619
Kendall Robust Correlation R_K, $n = 70$, $p \leq 0.025$									
$CC(\psi)$	0.586	0.544	0.462	0.595	0.577	0.623	0.300	0.686	0.448
$AND(\psi)$	0.289	0.308	0.286	0.343	0.334	0.341	0.200	0.356	0.241
$OR(\psi)$	0.553	0.596	0.575	0.629	0.629	0.563	0.404	0.654	0.507

Table 5.2 Pearson's, Spearman's and Kendall's correlation for size-based Z measures.

Measure	CL(M)	CLC(M)	CLCD(M)	CLCE(M)	CYC(M)	KNOTS(M)	FIN(M)	FOUT(M)	SI(M)
Pearson Correlation R_P, $n = 70$, $p \leq 0.028$									
$v_l'(\psi)$	0.789	0.676	0.382	0.764	0.743	0.793	0.262	0.813	0.264
$v_u'(\psi)$	0.787	0.661	0.358	0.758	0.739	0.794	0.259	0.808	0.262
$DU(\psi)$	0.799	0.642	0.285	0.777	0.736	0.817	0.279	0.833	0.282
Spearman's Rank Correlation R_S, $n = 70$, $p \leq 0.002$									
$v_l'(\psi)$	0.782	0.727	0.638	0.785	0.753	0.775	0.416	0.838	0.603
$v_u'(\psi)$	0.764	0.695	0.602	0.762	0.725	0.785	0.363	0.824	0.556
$DU(\psi)$	0.767	0.682	0.593	0.777	0.728	0.784	0.377	0.832	0.600
Kendall Robust Correlation R_K, $n = 70$, $p \leq 0.003$									
$v_l'(\psi)$	0.603	0.543	0.460	0.602	0.583	0.628	0.305	0.691	0.441
$v_u'(\psi)$	0.565	0.502	0.431	0.552	0.533	0.619	0.262	0.655	0.402
$DU(\psi)$	0.567	0.518	0.431	0.584	0.558	0.613	0.280	0.659	0.430

Table 5.3 Pearson's, Spearman's and Kendall's correlation for structure-based Z measures.

Semantics-based Correlation, $n = 70$											
		Pearson			Spearman			Kendall			
		$Cov(\psi)$	$O(\psi)$	$\chi(\psi)$	$Cov(\psi)$	$O(\psi)$	$\chi(\psi)$	$Cov(\psi)$	$O(\psi)$	$\chi(\psi)$
CL(M)	R	0.104	-.623	0.646	0.054	-.616	0.686	0.042	-.495	0.480
	p	**0.391**	0.000	0.000	**0.660**	0.000	0.000	**0.624**	0.000	0.000
CLC(M)	R	0.184	-.466	0.414	0.186	-.480	0.541	0.146	-.364	0.379
	p	**0.127**	0.000	0.000	**0.124**	0.000	0.000	**0.085**	0.000	0.000
CLCD(M)	R	0.229	-.215	0.116	0.243	-.369	0.433	0.190	-.280	0.310
	p	**0.057**	**0.075**	**0.340**	0.042	0.002	0.000	0.026	0.003	0.000
CLCE(M)	R	0.126	-.559	0.546	0.110	-.587	0.636	0.079	-.461	0.440
	p	**0.300**	0.000	0.000	**0.363**	0.000	0.000	**0.355**	0.000	0.000
CYC(M)	R	0.148	-.534	0.509	0.137	-.531	0.590	0.103	-.416	0.412
	p	**0.221**	0.000	0.000	**0.258**	0.000	0.000	**0.234**	0.000	0.000
KNOTS(M)	R	0.062	-.587	0.637	-.018	-.629	0.721	-.034	-.530	0.542
	p	**0.613**	0.000	0.000	**0.884**	0.000	0.000	**0.708**	0.000	0.000
FIN(M)	R	0.150	-.224	0.212	0.245	-.170	0.267	0.184	-.132	0.193
	p	**0.216**	**0.062**	**0.078**	0.041	**0.160**	0.026	0.034	**0.164**	0.026
FOUT(M)	R	0.096	-.572	0.582	0.046	-.599	0.722	-.008	-.479	0.541
	p	**0.430**	0.000	0.000	**0.704**	0.000	0.000	**0.930**	0.000	0.000
SI(M)	R	0.123	-.235	0.227	0.220	-.406	0.480	0.159	-.315	0.339
	p	**0.309**	0.050	**0.059**	**0.067**	0.000	0.000	**0.063**	0.001	0.000

Table 5.4 Pearson, Spearman and Kendall for semantics-based measures.

PART III

SPECIFICATION VISUALIZATION

CHAPTER 6

CROSSING THE BORDERLINE – FROM FORMAL TO SEMI-FORMAL SPECIFICATIONS

A. Bollin

IFIP Journal on Software Engineering Techniques: Design for Quality. Springer, 227:73–84, 2006.

Abstract

Being part of the systems' documentation state-based formal specifications can play a crucial role in the software development process. However, besides dense mathematical expressions, their semantical compactness and lack in visually appealing notations impede their use and comprehensibility among different stakeholders. One solution to this problem is to enrich the specification by a semi-formal view, in most cases diagrams with a sufficiently understood semantic meaning.

However, as control- and data-dependencies within declarative specifications are hard to detect, existing approaches only cover statics-bearing diagrams. As a way out this paper presents an approach for control- and data dependency analysis within declarative formal specifications. Based on these dependencies, UML diagrams showing static and dynamic properties of the specification are generated.

6.1 Introduction

Formal software specifications are usually recommended as means to produce high-quality software. They can solve the verification problem ("do the system right"). But, even if the

system has been refined correctly, there is another problem to be solved: the validation problem ("do the right system").

What sounds like a requirements elicitation problem also has to do with the problem of choosing a suitable notation. The risky part is that the stakeholders of the project (developers, customers, authorities) have to agree upon the meaning of the formal specification. Here, unclear requirements and specifications lead to futile validations easily. And this, in consequence, leads to a "buggy" system. So, the problem is not the formal notation, as its semantics is well-defined. The problem is the likely misinterpretation of concepts – due to the different habits of the stakeholders.

So, why not just combining formal specifications and wide-spread semi-formal approaches? Such a combination would have several advantages. Different views (either of graphical or textual/mathematical nature) convey the concepts much better between different stakeholders. The approaches are not meant to replace each other, but they extend the possibilities of concept description: properties of semi-formal descriptions get deducible (by stepping into the formal world) and formal specifications can be described at an even more abstract level.

Several research teams are working on the issue of mapping graphical approaches (e.g. UML) to formal specification languages. The generated specification is then used for consistency checking and test-data generation [15, 20]. Moving the other way round still has its limitations [10]. The reasons are the superficial analysis of state-based specification and missing flow of control. As a result only static class diagrams (that represent state variables of the specification) are generated so far.

With the reconstruction of control- and data- dependencies (via specification transformations [16]) much more gets possible: slices and chunks allow to excerpt pieces of the specification text with well-defined semantics, and cluster generation allows for carving out higher level specification concepts [4]. Finally, and as demonstrated in this contribution, by making use of control dependencies it gets feasible to visualize dynamic behavior of specifications the first time.

The contribution is structured as follows. Section 2 explains the need for bridging the gaps in more detail and presents the state-of-the-art of transforming UML diagrams to formal specifications and vice-versa. It gives special attention to some limitations of existing approaches: the scaling problem, and missing dynamics. Section 3 discusses ways to identify relevant elements within Z specifications [19] which will then be the basis for control- and data-dependency reconstruction. Section 4 then presents rules for the transformation (based on these dependencies). It explains the approach by making use of a small Z specification. The paper concludes with a short summary and an outlook.

6.2 Bridging the Gap

It would be the dream of every maintenance personnel, but neither does a SW-system, in general, adapt itself to changing situations or domains (retaining or even improving its quality), nor is it just straight-forward to produce new software products of high quality. As Glass [8, p.122] points out, comprehending the requirements and valid (and consistent) documentation is essential, but rarely all documents are available or are of suitable quality.

This is one of the main problems during software comprehension[1]. A situation that should be improved whenever possible.

6.2.1 Comprehension Challenges

There are several models describing how to maintain software systems [2, 17], but all of them stress that it is necessary to first comprehend the requirements *and* the relevant parts of the underlying system. Starting from scratch and stepping through the code is very time-consuming. Banker et. al. point out the fact that the time needed to comprehend a system on the basis of software code alone is about 3.5 times longer than comprehending the system by additionally studying its documentation [1]. Thus making use of design documents and specifications helps in saving time. But where are the problems?

Well, formal specifications are closer to the requirements – but comprehension needs special skills, and so stakeholders seem to shy away from them. The problems are manifold and are the result of the following gaps between the two worlds:

P1 Formal specifications are said to be of high perceived complexity.

P2 Not all stakeholders that are forced to comprehend the systems (and then to decide) are able to read and understand the underlying documents.

P3 Relating formal specifications with well defined semantics to less formal notations is a loss in precision.

P4 Creating formal specifications from less formal documents is impeded due to information deficiencies. It needs effort to fill the gaps.

What is easily overseen is that comprehending a system means the reconstruction of the missing documentation anyway. Concerning problem P1, the overall (and inherent) complexity cannot be reduced. But there are approaches to deal with the density of specifications [3]. And the remaining challenges P2 to P4 (understandability and formality) can be dealt with by consciously mapping the relevant representations to each other. The gaps can be bridged.

6.2.2 Impediments

The statements so far lead to one observation: the better and more extensive the documentation the easier the comprehension process. A less formal or less mathematical document can also serve its purpose. And a further improvement to the situation is to combine the formal and semi-formal documentation, to reconstruct parts of the documentation, and to be able to switch between different types of notation as needed.

As explained below, complexity (problem P1) does not make difficulties. In fact it is in the nature of formal specifications, and it concerns two aspects:

1. Usually there are too few clues for reconstructing the original structure. Putting too much structure into a formal specification is understood to be a hint towards implementation, something that is avoided at the time of writing the specification.

2. The declarative nature also impedes the reconstruction of the behavior. There is no execution-sequence, which is known from programming languages.

[1] According to Glass it is second only. The main cause for software comprehension problem is staff turnover.

Above all, the latter aspect is crucial. As there is no execution order and, in general, there are no control statements, control- and data-dependency are not predominant concepts. Well-known techniques from the field of program comprehension cannot be applied directly. A state-based specification focuses on, naturally, states, and alternative forms of representation are then restricted to just static information, too.

6.2.3 Related Work

For the reasons mentioned in Section 2.2 the mapping between formal specifications and less formal approaches is limited to static information. Besides formal extensions to existing notations (e.g. VDM-link to UML [5], or Petri-nets with Z extensions [9]), two directions of the mapping are possible.

Firstly, there is a mapping between some graphical notations to formal specifications. As UML is wide-spread, most of them take static UML diagrams and generate formal state descriptions of it (e.g. UML to Z [6, 11], or UML to Z++ [14, 18]). The approaches have in common that formal specification skeletons are generated which then have to be completed by the designer. As a second step the resulting predicates are simplified, leading, finally, to a full and compact formal specification. This means that semantics has to be added by the designer, but when dealt with it carefully, the specification can be taken to prove properties of the system. Results can then be mapped back to the design documents and deficiencies eliminated.

The other way round is the mapping of formal specification to some graphical notation. An early approach is the Z visualization of Kim [13], who makes use of constraint diagrams (a notation formally defined by Kent [12]). The notation is able to express predicate logic, but there is no integration into existing frameworks. And it is not UML, which does not really ease the understanding among some stakeholders.

However, not all the time the full content has to be imparted, and UML, keeping on spreading, is a good target for the transformation. The approach of Fekih et.al maps B specifications to UML [7]. It takes the state space of the specification and creates an UML class for every abstract set that is element in the domain of relations. The transformation rules are simple but lead to incomplete class diagrams as operations are not regarded. In addition to that the generated classes are not associated. Idani and Ledru improve the approach by mapping occurring relations to UML associations [10]. Furthermore they take operations into account and provide an algorithm for mapping an operation as a method to the most suitable class. Altogether this leads to a more complete static UML diagram.

Contributions mapping formal specifications to UML diagrams follow a pragmatic approach: sets do correspond to classes and relations do correspond to associations. However, as also noted in [10], the resulting diagrams provide less information than the formal specification, and dynamics is not touched at all.

6.3 Theoretical Background

The reconstruction of dependencies within declarative specification languages is not straightforward. When looking for control constructs (which will then be the basis for the reconstruction of dynamic behavior), one has to be careful about the basic elements the control is defined about.

Schema	Approximation via Conditions	Related Primes
S	$post\ S \rightrightarrows_c\ pre\ S$	$po_S \rightrightarrows_c pr_S$
$\neg\ S$	$post_s(\neg\ S) \rightrightarrows_c\ pre_w(\neg\ S)$	$po_S \rightrightarrows_c pr_S$
$S \vee T$	$post(S \vee T) \rightrightarrows_c\ pre(S \vee T)$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \Rightarrow T$	$post_s(S \Rightarrow T) \rightrightarrows_c\ pre_w(S \Rightarrow T)$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \wedge T$	$post(S \wedge T) \rightrightarrows_c\ pre_w(S \wedge T)$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \Leftrightarrow T$	$post_s(S \Leftrightarrow T) \rightrightarrows_c\ pre_w(S \Leftrightarrow T)$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \upharpoonright T$	$post(S \upharpoonright T) \rightrightarrows_c\ pre_w(S \upharpoonright T)$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \fatsemi T$	$post_s(S \fatsemi T) \rightrightarrows_c\ pre(S \fatsemi T)$	$(po_S \cup po_T) \rightrightarrows_c pr_S$
$S \gg T$	$post_s(S \gg T) \rightrightarrows_c\ pre(S \gg T)$	$(po_S \cup po_T) \rightrightarrows_c pr_S$

Table 6.1 The table summarizes the control dependencies occurring between pre- and postcondition primes in Z schema operations.

6.3.1 Specification Primes

Specifications are constructed from basic (atomic) units, called specification literals. They can be identified by looking at the grammar of the specification language. As an example, the Z predicate "$assigned \subseteq Permitted$" contains the specification literals "$Assigned$", "$\subseteq$", and "$Permitted$". Specification literals are not very expressive when standing alone. It is the combination of literals that makes them rich in content. By aggregating specification literals, *prime objects* of a specification are built.

Definition 19 *A specification prime object represents the basic entity of a specification. It is built out of specification literals and forms logic, syntactic, or semantic units.*

In specification languages prime objects can be expressions or predicates, but they can also be generic type or schema type definitions. When looking at the decoration of identifiers ($\boxed{'}$ or $\boxed{!}$ for after-states), post condition primes can easily be identified.

Definition 20 *A Z-specification prime p is considered a post-condition prime, if prime p contains an after state identifier. Otherwise it is considered a pre-condition prime.*

The following two predicates of the $AssignResource$ operation schema in the *Access-Control* Z Specification (see App. A) can be assigned to one pre- and one post-condition prime,

$$\begin{array}{ll} user? \notin \text{dom}\ Assigned & \%Precondition\ Prime \\ Assigned' = Assigned \cup \{user? \mapsto resource?\} & \%Postcondition\ Prime \end{array}$$

as the second prime contains an after-state identifier ($Assigned'$). The identification of control-dependencies is then based on these definitions.

6.3.2 Dependencies

The approach is based on the following simple idea: Preconditions determine whether predicates in the postcondition part are evaluated or not. Thus in specifications post-

Type	Symbol	A	B	Type	Symbol	A	B
Relation	$A \leftrightarrow B$	*	*	Part. Surj.	$A \twoheadrightarrow B$	1..*	0..1
Partial	$A \nrightarrow B$	*	0..1	Total Surj.	$A \rightarrow\!\!\!\!\!\rightarrow B$	1..*	1
Total	$A \rightarrow B$	*	1	Total Bij.	$A \rightarrowtail B$	1	1
Part. Inj.	$A \nrightarrow B$	0..1	0..1	Total Inj.	$A \rightarrowtail B$	0..1	1

Figure 6.1 According to [10], relations between sets A and B are mapped to associations with given multiplicities.

conditions are control dependent on pre-conditions, and the problem of the identification of control dependencies is reduced to the problem of the identification of pre- and post-conditions.

Not all specification languages do make pre- and post-conditions explicit. Additionally, when schema operations are used, pre- and post-conditions have to be calculated by performing a semantic analysis, a time- and resource-consuming task which cannot be fully automated. In [3] a syntactic approximation to the semantic analysis (see Tab. 6.1) is suggested and it is formally shown that this approximation yields suitable results.

Definition 21 *Let S be a schema of a Z specification. Furthermore, let pr_S be the set of pre-condition primes of S and po_S the set of post-condition primes of S. There is control dependency ($po_S \rightrightarrows_c pr_S$) within schema S, if pr_S and po_S are not empty.*

Table 6.1 summarizes the dependencies among the predicates in the schemas. When there is flow of control, data-dependencies are easily detected:

Definition 22 *A specification prime p is data dependent on a specification prime q ($p \rightrightarrows_d q$) if (i) there exists at least one identifier v (literal denoting a data element) that occurs in both p and q, and (ii) v is defined in q and used in p, and (iii) either p and q are in the same scope, or p is control dependent on q.*

6.4 Semi-Formal Transformation

With the reconstruction of dependencies within declarative specifications it gets possible to exploit both, static *and* dynamic diagrams of UML. In the following the Access Control system (see App. A) is taken to illustrate the transformation process.

6.4.1 Static Diagrams

The mapping to static class diagrams is based on the idea of looking for global type definitions. It follows mainly the approach of Idani et. al. [10], but omits assigning the operations to derived (and hard to find pertinent) classes. Instead it introduces a system class and assigns the operations to it.

Rule 1 *Every state schema corresponds to a root class. The root class is associated with the stereotype <<system>> and the name of the schema.*

Rule 2 *In a Z specification "given sets" correspond to classes in the UML specification. They are connected to the system class by using a "use" association.*

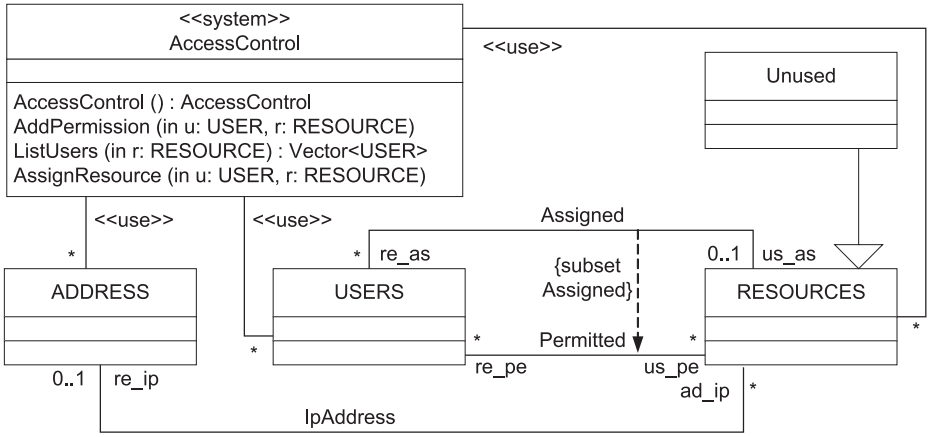


Figure 6.2 Applying Rules 1 to 6 leads to a static UML Diagram for the Access Control system specification (see Appendix A).

Typically this exactly is the view of a specification's designer: having a state and relevant operations. Abstract types are mapped to classes, and these classes are associated to the system class.

A specification also consists of several identifiers holding the state. When they describe relations between given sets, then they are resolved as associations. Subset relations ($\subset$ or $\subseteq$) are made explicitly. Finally, operation names are added, and for the description of the diagram unique names for the associations are introduced:

Rule 3 *Every variable representing relationships between entities in the state schema is translated to associations. It holds that (i) multiplicity is resolved by the mapping rules presented in Fig. 6.1, and (ii) subsets between relations are resolved by a subset constraint.*

Rule 4 *Every variable representing a subset of entities in the state schema is translated to a specialization class and connected to its root class.*

Rule 5 *Associations do get role-names. They are built by combining the first characters of the source class and association name.*

Rule 6 *Every operation schema is added as a method to those system root class, which has been included in the operations' declaration part. The initialization schema is mapped as a constructor to this class.*

Fig. 6.2 presents the result of the transformation. The system class contains the operations. As there are three given set definitions (Users, Resources, Addresses), three classes are introduced and connected to the system class. $Unused$ is modelled as a subset of $Resources$, and $Assigned$ and $Permitted$ are enriched by a subset constraint.

In fact, the above algorithm leads to class/attribute candidates. Still some problems with the transformation remain. It works well when there is only a small set of given sets. With larger specifications the approach of taking given sets as classes leads to a huge static UML diagram. Carving out higher-level concepts [4] and grouping them into extra diagrams might be a way out.

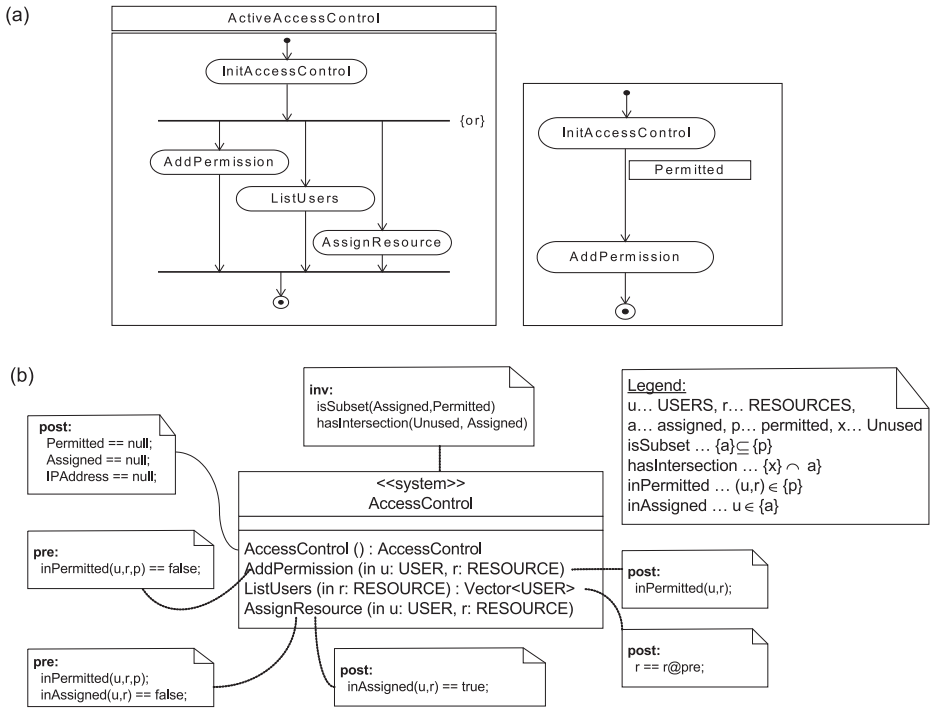


Figure 6.3 (a) Applying rules 7 and 8 leads to two UML Activity Diagrams, (b) applying rule 9 leads to an annotated static UML Diagram that makes use of OCL-like annotations.

6.4.2 Dynamics

UML also allows for dynamic diagrams, and as there is also some sort of dynamics within specifications it should be represented in a convenient way. The approach makes use of UML's activity diagrams and focusses on the above defined notion of control and data dependencies.

Rule 7 *Every schema operation is represented by an activity diagram. Logical operations are resolved as boolean expressions. Sequential operations are transformed to sequential control arcs.*

Rule 8 *Control dependencies between two operations are mapped to activity diagrams with control flow vertices. Data dependencies are mapped to activity diagrams by using object flow nodes with the label of the relevant identifier(s).*

The specification has one schema operation called $ActiveAccessControl$. It contains a sequential schema operation and the logical combination of the remaining operation schemas. Fig. 6.3(a), left side, presents the result of rule 7. It puts the initialization in sequence to the operations, which are logically combined by an OR-operation.

According to the definition of control dependency and the rules in Tab. 6.1, there is, for example, control dependency between the two schemas $AccessControl$ and $AddPermission$ (as $AddPermission$ includes the state schema and has post-condition primes which get dependent on the $AddPermission$'s pre-condition primes). Another diagram is added

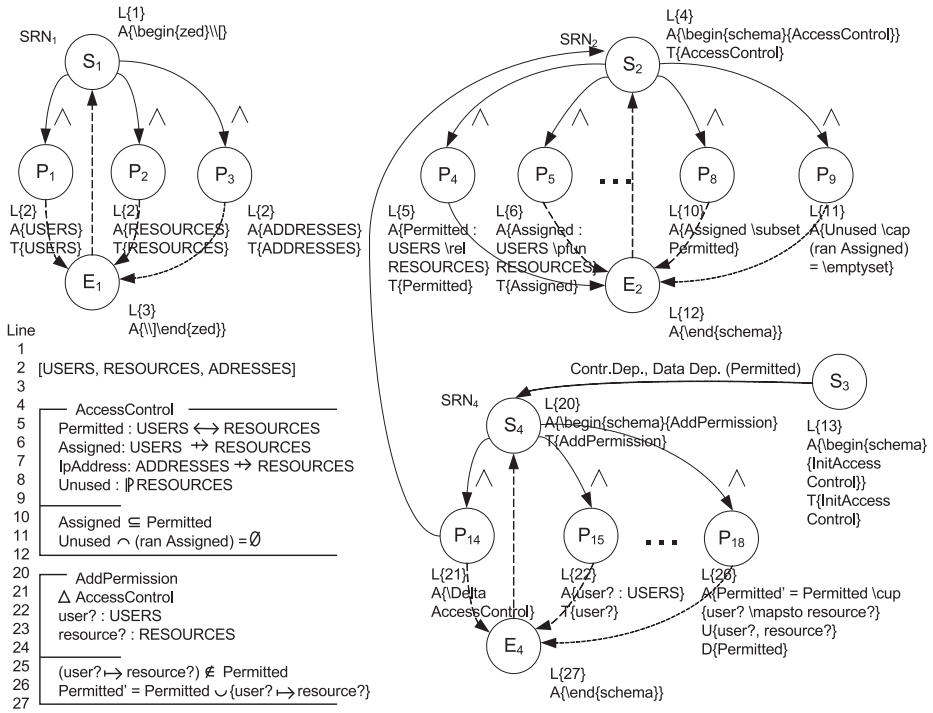


Figure 6.4 Parts of the ASRN net representation and source of the AccessControl specification. Primes are mapped to vertices and annotations are used for the transformation process.

and thus makes the control dependency explicit. Due to the fact that there is also data-dependency (the value of $Permitted$ is relevant), $Permitted$ is added as an object flow node. Finally, the more formal part of the specification must not be forgotten. By using OCL constraints, the relevant pre- and post-condition predicates can be included.

Rule 9 *Pre- and Post-conditions of operation schemas are mapped to pre and post OCL comments to the system class; the predicates of the state schema are mapped to the class as OCL invariants. For Z operations expressive function names are to be chosen, their semantics is to be explained in a legend box.*

Fig. 6.3(b) demonstrates the result of the transformation to a set of OCL comments. The constructor (initialization schema) gets a postcondition constraint, the three operation schemas get a pre- and post-condition each. The post-condition of $ListUsers$, e.g., tells us that the application does not change the set of resources ($r == r \bullet pre$, the after-state is the same as the before state). The system class is associated with the class invariant. Typically as is, predicate logic, Z functions, and operations are difficult to express in OCL. Here it is suggested to choose mnemonic names instead.

6.4.3 Automatic Generation

Specification primes are the basic, semantics-bearing elements of a specification. In [16, 3] it is described that state-based specifications can be mapped to a graph representation called ASRN (*Augmented Specification Relationship Net*), and that this graph can be used to detect dependencies by just breaking down the task to reachability conditions. This net is now used to simplify the transformation to UML. A small part of the AccessControl ASRN net can be found in Fig. 6.4.

The basic idea behind the net is that higher level primes are made up of a set of start and end vertices – which contain prime elements. Every vertex gets annotated (line numbers, text, and definition and use information of literals), and references and dependencies to other primes are expressed by (classified) edges between these primes.

The mapping process to UML is then straight-forward. Rules 1, 2, and 6 are based on the identification of state and operation schemas. Relevant start vertices have to contain type-declarations (`T{some_id}`). By looking at their successor vertices, operations and state schemas get separated (as only operations contain `Delta` and `Xi` annotations referencing other state schemas) and the corresponding UML diagrams can be generated. When all state schemas are identified, rules 3, 4, and 5 are resolved. Associations are resolved by looking at related prime vertices that contain type-declarations and relevant specification annotations (like $\leftrightarrow$ or $\nrightarrow$, according to Tab. 6.1).

The same approach is used for the generation of activity diagrams (rules 7 and 8). As the ASRN also contains control and data-dependency arcs, they are easily mapped to transitions in activity diagrams. Finally, for the generation of the OCL text (rule 9), again the annotations connected to the primes (`A(source)`) are parsed and printed.

Mapping the rules (on the basis of the ASRN) to a program is straight forward and the transformation can be done in reasonable time. However, the problem is still the neat visualization of all the diagrams - and for an optimal positioning of all the objects on the screen some user action is still needed.

6.5 Conclusion

Comprehension is an inevitable task during software maintenance and development phases, and specifications, when kept up-to-date, are valid sources. However, due to their complexity it is not surprising that formal specifications languages are said to be write-only languages.

This paper discusses ways in transforming formal Z specifications to UML in order to open the documents to a wider range of stakeholders. Existing approaches only cover static information, but state-based specifications also deal with state changes – and thus dynamics. In contrary to existing approaches (which the pure focus on class diagrams) this paper suggests to make also use of activity diagrams. It explains how to identify control dependencies, which are then the basis for the latter reconstruction of dynamic behavior within declarative specifications.

There are still some limitations that should not be concealed. As with other approaches the issue of scalability is not solved, and in addition to that it is still hard to decide whether a class candidate is a pertinent class or not. To test the applicability of the approach a framework for dealing with large Z specifications (combining slicing, chunking, and UML transformation) is under further development.

The approach does, by far, not lead to a perfect UML representation of the specification. But it provides a good picture of *what* is *in* the specification. In fact it can at least be used to speed up the re-construction of concepts behind. And as dynamics is at least as important as statics, this approach should be a step further into the direction of a more useable framework and increase the use of formal specification.

REFERENCES

[1] Rajiv D. Banker, Gordon B. Davis, and Sandra A. Slaughter. Software development practices, software complexity, and software maintenance performance: A field study. In *Management Science*, volume 44, pages 433–450. Inst. for Operations Research and the Management Sciences, April 1998.

[2] R. V. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Software*, 7(1):19–25, 1990.

[3] Andreas Bollin. *Specification Comprehension – Reducing the Complexity of Specifications*. PhD thesis, University of Klagenfurt, 2004.

[4] Andreas Bollin. Maintaining formal specifications. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary*, pages 442–453, 2005.

[5] Jeremy Dick and Jerome Loubersac. A visual approach to VDM: Entity-structure diagrams. Technical Report DE/DRPA/91001, Bull, 68, Route de Versailles, 78430 Louveciennes (France), 1991.

[6] Sophie Dupuy, Yves Ledru, and Monique Chabre-Peccoud. An overview of RoZ: A tool for integrating UML and Z specifications. In *Proceedings of CAiSE'00*, pages 417–430, 2000.

[7] Houda Fekih, Leila Jemni, and Stephan Merz. Transformation des spècifications B en des diagrammes UML. In *Proceedings of Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL'*04, pages 131–148, 2004.

[8] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2003.

[9] Xudong He. PZ Nets - a formal method integrating Petri Nets with Z. *Information and Software Technology*, 43(1):1–18, 2001.

[10] Akram Idani and Yves Ledru. Object oriented concepts identification from formal B specifications. In *Formal Methods in Industrial Critical Applications, FMICS'04*, 2004.

[11] Akram Idani, Yves Ledru, and Didier Bert. Derivation of UML class diagrams as static views of formal B developments. In *7th International Conference on Formal Engineering Methods, ICFEM 2005*, pages 37–51, 2005.

[12] Stuart Kent. Constraint diagrams: Visualising invariants in object-oriented models. In *In Proceedings of OOPSLA'97*. ACM Press, 1997.

[13] Soon-Kyeong Kim and David Carrington. Visualization of formal specifications. In *In Proceedings Sixth Asia Pacific Software Engineering Conference (ASPEC'99)*, pages 102–109. IEEE Computer. Society Press, Los Alamitos, CA, USA, 1999.

[14] Soon-Kyeong Kim and David Carrington. A formal mapping between UML models and Object-Z specifications. *Lecture Notes in Computer Science*, 1878:2–21, 2000.

[15] Règine Laleau and Amel Mammar. An overview of a method and its support tool for generating B specifications from uml notations. In *Fiftheenth IEEE Conference on Automated Software Engineering*, 2000.

[16] Roland T. Mittermeir and Andreas Bollin. Demand-driven specification partitioning. In *Proceedings of the 5th Joint Modular Languages Conference, JMLC'03*, Ausgust 2003.

[17] Helfried Pirker. *Specification based Software Maintenance (a Motivation for Service Channels)*. PhD thesis, University of Klagenfurt, 2001.

[18] D. Roe, K. Broda, and A. Russo. Mapping UML models incorporating OCL constraints into Object-Z. Technical Report ISBN/ISSN: 1469-4174, Imperial College of Science, Technology and Medicine, Department of Computing, 2003.

[19] J.M. Spivey. *The Z Notation*. C.A.R. Hoare Series. Prentice Hall, 1989.

[20] Ninh-Thuan Truong and Jeanine Souquieres. An approach for the verification of UML models using B. In *Proceedings of the 11th IEEE Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)*, 2004.

Appendix A - Access Control Specification

$$[USERS, RESOURCES, ADDRESSES]$$

AccessControl

$$\begin{aligned} &Permitted : USERS \leftrightarrow RESOURCES \\ &Assigned : USERS \nrightarrow RESOURCES \\ &IpAddress : ADDRESSES \nrightarrow RESOURCES \\ &Unused : \mathbb{P}\, RESOURCES \\ \hline &Assigned \subseteq Permitted \\ &Unused \cap (\text{ran}\, Assigned) = \varnothing \end{aligned}$$

InitAccessControl

$$\begin{aligned} &AccessControl \\ \hline &Permitted = \varnothing \\ &Assigned = \varnothing \\ &IpAddress = \varnothing \end{aligned}$$

AddPermission

$$\begin{aligned} &\Delta AccessControl \\ &user? : USERS \\ &resource? : RESOURCES \\ \hline &(user? \mapsto resource?) \notin Permitted \\ &Permitted' = Permitted \cup \{user? \mapsto resource?\} \end{aligned}$$

ListUsers

$$\begin{aligned} &\Xi AccessControl \\ &resource? : RESOURCE \\ &st! : \mathbb{P}\, USERS \\ \hline &st! = \text{dom}(Permitted \rhd \{resource?\}) \end{aligned}$$

$$
\begin{array}{|l}
\hline
\textit{AssignResource} \\
\Delta \textit{AccessControl} \\
\textit{user}? : \textit{USERS} \\
\textit{resource}? : \textit{RESOURCES} \\
\hline
(\textit{user}? \mapsto \textit{resource}?) \in \textit{Permitted} \\
\textit{user}? \notin \mathrm{dom}\ \textit{Assigned} \\
\textit{Assigned}' = \textit{Assigned} \cup \{\textit{user}? \mapsto \textit{resource}?\} \\
\hline
\end{array}
$$

$$
\textit{ActiveAccessControl} == \textit{InitAccessControl} \fatsemi (\textit{AddPermission} \lor \textit{ListUsers} \lor \textit{AssignResource})
$$

CHAPTER 7

COUPLING-BASED TRANSFORMATIONS OF Z SPECIFICATIONS INTO UML DIAGRAMS

A. Bollin

International NASA Journal on Innovations in Systems and Software Engineering, 7(4):283-292, 2011.

Abstract

Due to their accuracy in describing systems, formal specifications can play an important role during forward as well as reverse engineering activities. However, besides dense mathematical expressions, their lack in visually appealing notations impedes their use and exchange among different stakeholders. One solution to this problem is to enrich the specification by other views, in most cases UML diagrams. But the mapping is not trivial, and existing approaches have their impediments, among them the assignment of methods to classes – which has to be re-done by hand quite often.

By the example of Z, this paper demonstrates that the situation can be improved. The new approach combines existing mapping strategies, but additionally lets the assignment of methods rest on quality-related measures. The basic idea is to balance the values of coupling for all methods within and between the UML classes. With that, two issues are addressed: firstly, the mapping of sets, types, and operations (to UML classes and UML methods) is based on reproducible measures that are intuitively comprehensible. Secondly, implementations based on the resulting UML class diagrams very likely also have comparable quality-related properties.

7.1 Introduction

Today's systems become more and more software-intensive which typically means that software is the major component that provides the needed functionality. With that, reliability and dependability considerations gain in importance, and, by following the arguments in [14, p.55], formal methods hence lend themselves back to the software engineering community. Several stories of success and myths are around (and might balance each other [11, 5, 28]), but when used appropriately, they form the basis for ongoing software engineering steps like code- and test-case generation.

But there is a drawback. Even if the system has been refined correctly, there is another issue to be solved: the validation problem. What sounds like a requirements elicitation problem also has to do with the question of choosing a suitable notation (as several stakeholders in the project will have to agree upon the specification). Here, the mathematically dense notations might be an impediment. A pragmatic solution is therefore to combine formal specifications and other (semi-formal) graphical notations and thus to provide an additional view onto the specification.

As illustrated in Section 7.2.1, there are already approaches dealing with the mapping between graphical notations (like UML) and formal specifications. However, these transformations are not without restrictions. Formal specifications are usually not object-oriented and they are not necessarily concerned with classes. This contribution makes an attempt to resolve one of the stumbling blocks in the mapping of specifications to UML diagrams: that of a "pragmatic" layout of the diagrams. But what is the exact problem?

Well, apart from a possible loss in semantic expressiveness, the resulting diagrams quite often have to be re-formatted by hand so that developers and managers are satisfied with them and perceive them as useful. Especially the assignment of operations to classes is a problem that has to be resolved by human beings. Existing solutions do their best in finding pertinent classes for methods, but the mapping is not necessarily recognized as useful, and when the specification scales, classes with unbalanced design might be created.

Here, by the example of Z specifications [29], the paper suggests an alternative way in finding an improved mapping. The basic idea is to focus on the values of coupling between the operations in the specification and to use this information to find class assignments that are "optimal". Basically, the objective is to find a mapping such that the coupling between the methods within a class is at a maximum, and the coupling to methods in other classes is at a minimum. The optimization process will be discussed in more details in Section 7.4.

The contribution is structured as follows. Section 2 explains the need for mapping strategies in more detail and presents approaches transforming UML diagrams to formal specifications and vice-versa. It also gives attention to some limitations of existing approaches. Section 3 discusses the transformation process for Z specifications. It also provides the necessary background for the calculation of slice-based coupling values. Section 4 then explains the approach by making use of a small Z specification. Finally, the paper concludes with a short summary and an outlook.

7.2 Formal Specification Transformations

The idea of combining formal specifications with other notations is not new, and the existing approaches help to focus on orthogonal properties of the underlying system. Understandability is gained due to the different views, and decisions are alleviated. Dick argues

Type	Symbol	A	B
Relation	$A \leftrightarrow B$	*	*
Partial	$A \nrightarrow B$	*	$0\,..\,1$
Total	$A \rightarrow B$	*	1
Part. Inj.	$A \rightarrowtail B$	$0\,..\,1$	$0\,..\,1$
Part. Surj.	$A \twoheadrightarrow B$	$1\,..\,*$	$0\,..\,1$
Total Surj.	$A \twoheadrightarrow B$	$1\,..\,*$	1
Total Bij.	$A \rightarrowtail\!\!\!\!\twoheadrightarrow B$	1	1
Total Inj.	$A \rightarrowtail B$	$0\,..\,1$	1

Table 7.1 As defined in [16], relations between sets A and B are mapped to associations with the given multiplicities.

in [8] that confidence and acceptability is raised and changes of the system are alleviated. As the transformation is possible in two directions, also some weaknesses of purely graphical notations can be eliminated. The following section summarizes existing approaches briefly and then moves on to the issue of the improved mapping strategy.

7.2.1 Related Work

Besides formal extensions to existing graphical notations (e.g. Petri-nets with Z extensions [13] or VDM-link to UML [7]), two classes of approaches for specification transformations are existing.

The first class comprises approaches that map graphical notations to formal specifications. UML is wide-spread, so most of them take static UML diagrams and generate formal state descriptions from it (e.g. UML to Z [9, 17], or UML to Z++ [20, 27]). The approaches have in common that formal specification skeletons are generated which then have to be completed by the designer/developer. After completion the resulting predicates are simplified, resulting in a compact formal specification. So, semantics has to be added by the designer, but the specification can then be taken to prove properties of the system and the results can then be mapped back to the design documents in order to eliminate deficiencies.

The second class comprises approaches that map formal specification to some graphical notation. One early approach is the visualization of Z defined by Kim [19], who makes use of constraint diagrams [18]. The notation is able to express predicate logic, but, unfortunately, there is no integration into existing frameworks. In addition, constraint diagrams look differently from UML diagrams, and so the understanding among different stakeholders is impeded again.

When not the whole semantics of a specification at hand has to be mapped, then UML, being state-of-the-practice, is a possible candidate. With the involvement of members of

the precise UML group[1] in the standardization process, it also gets an interesting target for the transformation process. The approach of Fekih et.al maps B specifications to UML [10]. It takes the state space of the specification and creates an UML class for every abstract set that is element in the domain of relations. The transformation rules are simple and lead to incomplete class diagrams as operations are not regarded. In addition to that the generated classes are not associated. Idani and Ledru improve the approach by mapping occurring relations to UML associations [16] (as summarized in Table 7.1). Furthermore they take operations into account and provide an algorithm for mapping an operation as a method to the most suitable class (called pertinent class). Altogether this leads to a more complete static UML diagram, though their approach neglects the dynamics behind operations.

In [2] the approach of Idani and Ledru is mapped to Z and extended by rules to cover also activity diagrams. This is done by regarding control and data dependencies that have been recalculated beforehand (via a specification transformation that is explained in more details in [24]). The approach has been refined and integrated into a Java-based environment called ViZ by Lessacher in 2007 [21]. Nevertheless, also this set of rules has its limitations as the mapping of operations is handled in such a way that all operations are part of a root class with the stereotype "$\langle\langle$system$\rangle\rangle$" (as will be shown in Section 7.3.2).

7.2.2 Mapping Strategies

Algorithms that map formal specifications to UML diagrams follow a pragmatic approach: sets correspond to classes and relations correspond to associations. This mapping is quite natural as programmers often use classes to represent abstract types. Their interrelation is then expressed by various forms of associations between them. The mapping of operations to classes is more sophisticated. One way in dealing with the situation is to take a look at the number of uses (and references) of these sets of types in the operations and to assign them to the most frequently used classes (as done by Idani and Ledru in [16] and extended later on in [15]). A contrary approach (as used in [2]) is to collect all of the operations and to put them into a separate class. However, both strategies have drawbacks:

- For the first approach it might happen that more than one class is pertinent for an operation. It is then up to the user to decide where to put the method to. While this is not a problem for the second approach, putting all operations into one class definitely does not scale-up very well.

- Besides abstract types, also the state-space is relevant. The first approach does not deal with this information. The second approach creates a system class for every state, but whenever several state spaces are included it is again not defined where a method has to be mapped to.

- None of the transformation rules do take implementation related issues into account. While implementation details are not an issue for formal specifications, this point gets important when (a) communicating them to different stakeholders and (b) using them as the basis for the ongoing development process.

[1]The precise UML group, created in 1997, tries to bring international researchers and practitioners together in order to develop the Unified Modelling Language (UML) as a well defined modeling language. See http://www.cs.york.ac.uk/puml/index.html for more details.

Measure	*Definition*
Inter-Schema Flow $\digamma(\psi_s, \psi_d)$ measures the number of primes of the slices in ψ_d that are in ψ_s	$\frac{\lvert(SU(\psi_d) \cap \psi_s)\rvert}{\lvert\psi_s\rvert}$
Inter-Schema Coupling $C(\psi_s, \psi_d)$ computes the normalized ratio of the flow $\digamma$ in both directions	$\frac{\digamma(\psi_s,\psi_d)\ \lvert\psi_s\rvert + \digamma(\psi_d,\psi_s)\ \lvert\psi_d\rvert}{\lvert\psi_s\rvert + \lvert\psi_d\rvert}$
Schema Coupling $\chi(\psi_i)$ is the weighted measure of Inter-Schema Coupling C of ψ_i and all n other schemata	$\frac{\sum_{j=1}^{n} C(\psi_i,\psi_j)\ \lvert\psi_j\rvert}{\sum_{j=1}^{n} \lvert\psi_j\rvert}$

Table 7.2 Slice-based measures for Inter-Schema Flow and Coupling as introduced for Z specifications in [4].

An improved mapping strategy should take these issues into account, and especially the third aspect can be used to improve the existing mapping strategies. When a specification is the basis for an implementation (especially when specifications are refined) then it is very likely that the dependencies between operations are still prevalent and affect implementation-specific properties. This includes the dual properties of coupling and cohesion and operations should be created in such a way that the values for coupling and cohesion are minimized (maximized) whenever possible.

The idea now is quite simple: one can use specification measures to decide where to map operations to. In [4] it is shown that slice-profiles [26, 23, 30] can be computed for Z specifications and that they can then be used to calculate specification-based coupling and cohesion measures. It is also demonstrated that the behavior of these measures can be compared to their corresponding measures in the field of ordinary programming languages. So, based on these values, it is suggested to add another rule set that regards the average values of coupling for every operation. Whenever necessary, it moves the operations to other (pertinent) classes in such a way that at the end of this process all values have reached an optimum (which means high values for coupling between methods in one class and lower values for coupling between methods of different classes).

Section 7.3 now introduces the necessary background for the calculation of slice-based coupling measures and the refined set of transformation rules.

7.3 Slice-based Transformation Process

The calculation of slice-based measures goes back to the notion of a static specification slice as introduced by Oda and Araki [25] and Chang and Richardson [6]. Their idea is to look for predicates that are part of pre- and postconditions and to introduce "control" dependencies between them. Their idea has been refined and extended by Bollin [1] which also led to the development of an environment called *ViZ* that now supports reverse engineering of formal Z specifications by slicing, chunking and clustering techniques [3].

Nr.	*Rule*
1	Every section in a Z specification corresponds to one UML class diagram.
2	Every state schema corresponds to a root class with the stereotype ⟪system⟫ and the name of the schema.
3*	Every given set A corresponds to a class in the UML specification, getting the name of the basic type.
4	Every inclusion of a state A in the declaration part of a schema B corresponds to an aggregation of the classes A and B (where B is the whole class).
5	Every use of a given set or free-type A in a state schema B corresponds to a ⟪use⟫ association between the classes A and B and with multiplicities $(*, 1)$.
6*	Every variable representing relationships between entities in a state schema is translated to associations. It holds that (i) multiplicity is resolved by the mapping rules presented in Table 7.1, (ii) subsets between relations are resolved by a subset constraint, and (iii) an identifier A representing a set of a type B is resolved by a generalization between class A and super-class B. Associations do get role-names. They are built by combining the first characters of the source class and association name.
7	Every use of a given set or free-type A in an operation schema B that has been assigned to a class C is mapped to a ⟪use⟫ association with multiplicities $(*, 1)$ between A and C.
8	Every identifier A in the declaration part of an operation schema is mapped to a parameter of the corresponding method B, annotated by "In" for input and "Out" for output. When the output is a set, then a Vector of the type is returned.
9	Every operation schema A is added as a method to those system root class which has been included in the operations declaration part. When there are several root classes possible, then A is added to a system class called "Operations". In this case a ⟪use⟫ association with multiplicities $(1, 1)$ is introduced between these classes. The initialization schema is mapped as a constructor to its corresponding system class.

Table 7.3 (Part I) Z mapping rules for the static part of a specification as defined by Bollin and Lessacher [2, 21]. Rules following the approach of Idani and Ledru [17] are marked by an asterisk.

7.3.1 Slice-based Coupling

For the definition of the measure of coupling we first need to introduce the notion of specification slice profiles and the union of all its slices. The basic idea is quite simple: for every post-condition prime in a schema ψ one has to calculate the corresponding slices. The set of all possible specification slices is called *Slice Profile* ($SP(\psi)$). The union of all slices in $SP(\psi)$ is called *Slice Union* ($SU(\psi)$).

The calculation of coupling follows the definitions to be found in [12]. First, an Inter-Schema Flow F is specified. It describes how many primes of the slices in the slice union are outside of the schema. Inter-Schema Coupling C is then computed by the normalized ratio of this flow in both directions. Finally, Schema Coupling χ is calculated by considering the Inter-Schema Coupling values to all other schemata. The definitions of the measures are summarized in Table 7.2.

Nr.	*Rule*
10	Every free-type corresponds to a UML class with the stereotype ⟪datatype⟫. Every constant of a free-type A is mapped to an attribute of the corresponding UML class. Every dataset of the constructor of a free-type A corresponds to an instance variable in the corresponding class A. Every link-set of the constructor of a free-type A corresponds to a recursive association of the corresponding class A with the name of the constructor and multiplicities $(0\,..\,1, 1)$.
11	Every global constant A representing a subset of a free-type B is mapped to a UML class with the name of A and the stereotype ⟪datatype⟫. Additionally, a generalization between the classes A and B is introduced. Every element of a subset A corresponds to a class attribute of class A and gets the name of the element and the type of A.
12	Every identifier in the declaration part of a schema A representing a sequence of a state schema or a type B is mapped to an association between classes A and B. For non-empty sequences the multiplicities are $(0\,..\,1, 1\,..\,*)$, otherwise $(0\,..\,1, *)$.
13	Every identifier in the declaration part of a schema representing a relation between a type A and a sequence of type B corresponds to an association between classes A and B. Multiplicity is resolved by the mapping rules presented in Table 7.3.2 where the multiplicity for class B is $1\,..\,*$ for non-empty sequences, and $*$ otherwise.

Table 7.4 (Part II) Extended set of rules (compared to the rule-set presented in [2] and [21]) for the static part of a Z specification, now also dealing with sequences and free-types.

7.3.2 Transformation Rules

The mapping of Z specifications to static class diagrams as introduced in [2] is based on the idea of Idani and Ledru [17]. However, the approach omits assigning the operations to derived classes. Instead, it introduces one or several system classes and assigns the operations to them. When an operation can be assigned to several system classes, then a helper class called "Operations" is created and the operation is assigned to it.

Experiments with larger specifications showed that free-types and sequences are used quite often. As they are not covered by the rules defined in [2], the existing set of rules had to be extended again. The resulting mapping strategy[2] can be found in Tables 7.3 and 7.4.

Fig. 7.1 presents the result for the transformation of the "Access Control Specification" as done by the *ViZ* environment[3]. *ViZ* generates an *XML* file that can be imported by UML modeling tools like *VisualParadigm*[4]. According to the mapping rules the system class contains all operations and the initialization schema as a constructor. As there are three given set definitions (called $USERS$, $RESOURCES$, and $ADDRESSES$), three classes are introduced and connected to this system class. Additionally, the identifier $Unused$ is modeled as a subset of $Resources$, and the associations $Assigned$ and $Permitted$ are enriched by a subset constraint.

[2] A description of the mapping rules (also covering activity diagrams) including Java-like pseudo-code can be found in [21].
[3] The specification is also used by other authors to demonstrate their mapping strategies and has thus been selected for this contribution.
[4] Visual Paradigm is part of the Visual Paradigm Suite and is free for academic sites. For more information see http://www.visual-paradigm.com. Page last visited: August 2011.

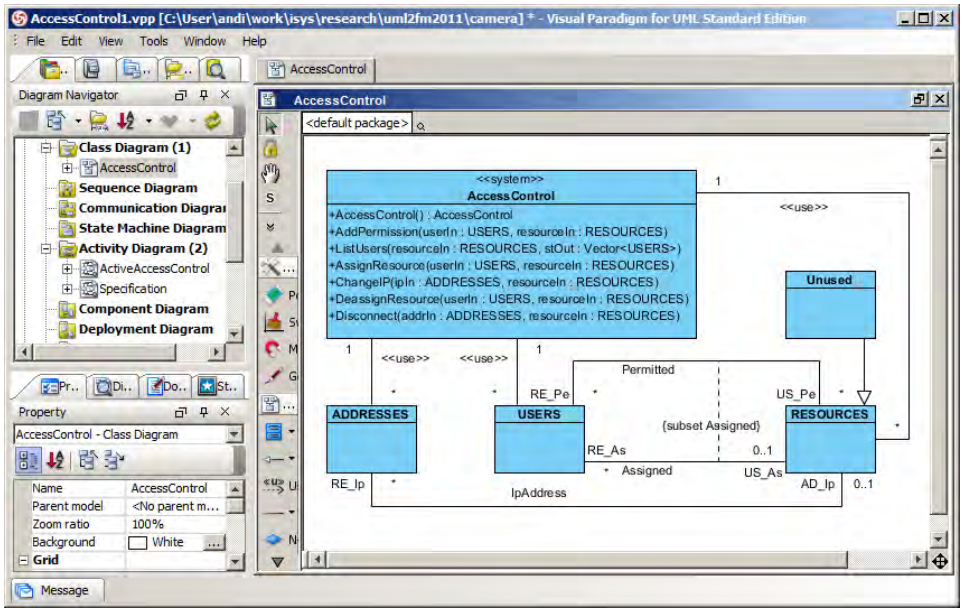


Figure 7.1 Applying rules 1 to 13 leads to a static UML Diagram for the Access Control system specification (see Appendix). The UML diagram has been exported by the *ViZ* environment and imported by Visual Paradigm 8.0 (and making use of the "Auto Layout" algorithm).

$C(\psi_s, \psi_d)$	S1	S2	S3	S4	S5	S6	S7	S8
$S1_{AccessControl}$	1.000	0.250	0.444	0.800	0.400	0.364	0.400	0.400
$S2_{InitAccessControl}$	0.250	1.000	0.154	0.222	0.143	0.133	0.143	0.143
$S3_{AddPermission}$	0.444	0.154	1.000	0.400	0.267	0.250	0.267	0.267
$S4_{ListUsers}$	0.800	0.222	0.400	1.000	0.364	0.333	0.364	0.364
$S5_{AssignResource}$	0.400	0.143	0.267	0.364	1.000	0.235	0.250	0.250
$S6_{ChangeIP}$	0.364	0.133	0.250	0.333	0.235	1.000	0.235	0.235
$S7_{DeassignResource}$	0.400	0.143	0.267	0.364	0.250	0.235	1.000	0.125
$S8_{Disconnect}$	0.400	0.143	0.267	0.364	0.250	0.235	0.125	1.000
$\chi(\psi_i)$	0.383	0.215	0.248	0.343	0.233	0.220	0.233	0.233

Table 7.5 Values for Inter-Schema Coupling C and schema coupling $\chi(\psi_i)$ for the Z schemata of the *Access Control specification*.

In fact, the set of rules operates well when there is only a small number of given sets and a few operations. With larger specifications that contain a lot of operations the approach of taking given sets as classes and putting them into the system class (as can be seen in Fig. 7.1) leads to huge static UML diagrams no developer would generate by hand. However, making use of coupling measures mitigates this situation. It also leads to an assignment of operations to classes such that the values for coupling within a class are maximized and the values for coupling to other classes are minimized.

7.4 Coupling-based Mapping

Contrary to existing approaches where pertinent classes are identified by the number of use or references to identifiers, the strategy for identifying pertinent classes is now based on the analysis of the values of Inter-Schema Coupling C. The objective is to "optimize" their values for the whole specification:

Definition 23 *The value for Inter-Schema Coupling of the transformed specification is optimal, when there is no other assignment of operations to classes such that the average value of all Inter-Schema Coupling values within the classes can be increased.*

When taking a look at the values of schema coupling $\chi(\psi_i)$ for the Access Control specification in Table 7.5 then we can see that the values are quite evenly distributed. Nevertheless, there are variations. The state space $AccessControl$ has the highest values as it is connected to all the other schemas, and $InitAccessControl$ has the lowest relation to all the other schemas. For an optimal distribution (where the values for coupling are maximized for methods within classes) the following additional rules are pursued:

Rule 14 *For every system class select all operations that have an Inter-Schema Coupling value lower than a given threshold ρ, and for every selected operation determine the set of possible (pertinent) class candidates.*

Rule 15 *Move every operation to one class-candidate by the following strategy: (a) select a class from the set of candidates that has no method at all, and (ii) if there is no such class select a class so that the average value for Inter-Schema Coupling for all class candidates stays at a maximum.*

Rule 16 *Look at every class and check whether a swap or delegation of methods between two classes increases the average values for Inter-Schema Coupling. If so, then delegate or swap the methods. Continue the steps till there are no more improvements.*

The problem of finding the optimal solution can be compared to the knapsack problem [22]. However, we know a bit about the preferred properties of the resulting classes and the optimal solution is approximated by the rules due to a simple strategy: at first it tries to avoid empty classes (the value for inter-class coupling would be zero). The first class-candidate that matches gets the method assigned to it, yielding a value for intra-schema coupling equal to 1 (per default). Secondly, when there is no class-candidate that is empty, then the method is assigned to one of the classes in such a way that the average value for schema coupling does not decrease too much. The last step is to try to find a swapping of the methods such that the average value for coupling increases a bit.

With the suggested procedure, one is able to calculate a balanced distribution of the operation schemas to the classes. In fact, the first couple of steps are quite straight forward and efficient. Choosing between all operations according to a given threshold can be done in linear time (when the Inter-Schema Coupling matrix is already given). The identification of the set of class candidates per method can also be conducted in one run when parsing the parameter lists of the methods. The same holds for the next step, the assignment of the methods to the classes. Empty classes are selected easily, and only in the case of already occupied classes one has to take a look at the matrix again and compute the average of the related Inter-Schema Coupling values.

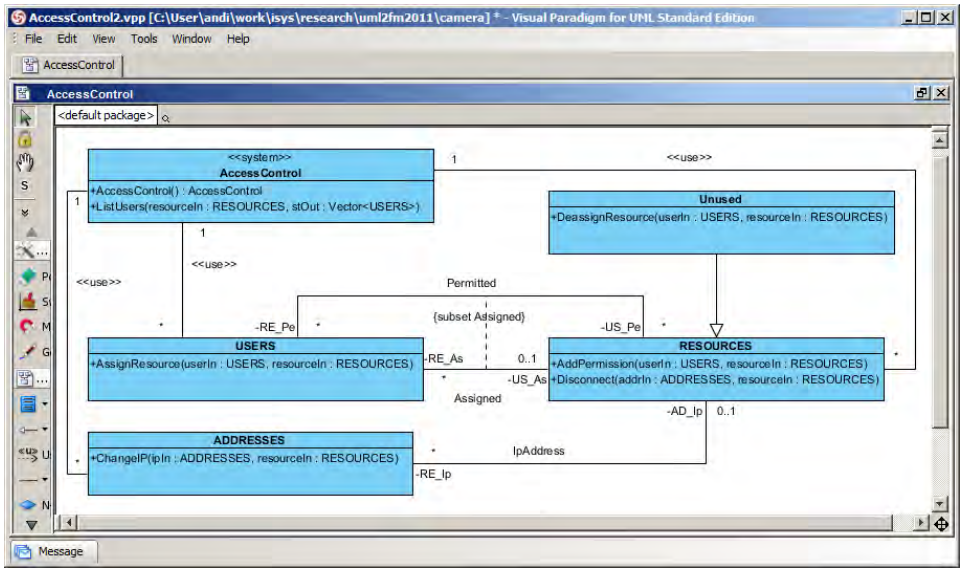


Figure 7.2 Applying rules 14 to 16 leads to a more balanced static UML Diagram for the Access Control system specification (see Appendix). Please note that the annotations containing the predicates, even though exported by the *ViZ* environment, have been omitted for reasons of readability.

Definitely more complex is the implementation of the last two steps, the delegation and swapping of methods. In the worst case, all possible combinations of methods per class have to be considered till the algorithm stops. In fact, it might be that there is more than one optimal solution[5], but the algorithm stops when no further improvement of the average value of the Inter-Schema Coupling values can be found.

This process of assigning methods to classes – according to a balanced distribution of Inter-Schema Coupling values – is now demonstrated by the example of the Access Control specification (see Appendix for the Z specification).

7.4.1 Example

The starting point of our example is the UML layout as presented in Fig. 7.1 (so rules 1 to 13 have already been applied). There, all the methods are element of one system class. Rule 14 now tells us that (for a given value ρ) we have to decide upon the methods we want to delegate to associated classes. For ρ let us assume to start with a value of 0.7, which means that we want to delegate those methods that have a value of Inter-Schema Coupling lower that 0.7. The relevant methods (see Table 7.5, rows $C(\psi_s, \psi_d)$ and $S1$) are therefore: $AddPermission$, $AssignResource$, $ChangeIP$, $DeassignResource$, and $Disconnect$. ($ListUsers$ has a value $C(ListUsers) = 0.800$ within the system class, so a value higher than ρ and it is skipped. The initialization schema is also skipped as it becomes the constructor of the system class). For the selected methods we have to calcu-

[5]This might be the case as the initial distribution of methods to empty classes is non-deterministic and the values of Inter-Schema Coupling between some of the operation schemas can be the same.

late the class-candidates (so the pertinent classes that are associated with the system class and that are referred to or used in the methods). The class-candidates and their pertinent classes are as follows:

- $AddPermission \rightarrow USERS$ or $RESOURCES$.
- $AssignResource \rightarrow USERS$ or $RESOURCES$.
- $ChangeIP \rightarrow ADDRESS$ or $RESOURCES$.
- $DeassignResource \rightarrow USERS$, $RESOURCES$, or $Unused$.
- $Disconnect \rightarrow ADDRESS$, $RESOURCES$, or $Unused$.

We now apply rule 15 to the specification. In this first step we are taking a look at the methods one by one. At first, $AddPermission$ could be delegated to either $USERS$ or $RESOURCES$ (as both of the classes are still empty). In our case let us take the $USERS$ class, which also means that the value for coupling within the USERS class increases from 0 to 1. Next is $AssignResource$ which goes to the $RESOURCES$ class as $USERS$ already contains one method. $ChangeIP$ is moved to $ADDRESSES$, $DeassignResource$ is moved to class $Unused$. For the $Disconnect$ operation we have the choice between a delegation to three classes (either $RESOURCES$, $ADDRESSES$, or $Unused$) that already contain operations. Thus the value of C has to be calculated for these three options:

- $C(Disconnect, AssignResource) = 0.250$.
- $C(Disconnect, ChangeIP) = 0.235$.
- $C(Disconnect, DeassignResource) = 0.125$.

In order to keep coupling for the classes at a maximum, $Disconnect$ is moved to $RESOURCES$ as a second method (as it already contains $AssignResource$). After applying rule number 15, the assignment of methods to classes looks as follows:

- $USERS$ contains $\{AddPermission\}$.
- $RESOURCES$ contains $\{AssignResource, Disconnect\}$.
- $ADDRESSES$ contains $\{ChangeIP\}$.
- $Unused$ contains $\{DeassignResource\}$.

We are now able to apply rule 16 to the UML diagram. At first, we try to swap the methods so that we manage to increase the average values of coupling. Swapping and delegation only makes sense for classes that contain more than one method, so we are looking at the $RESOURCE$ class.

- The method $AssignResource$ could be swapped with $AddPermission$ of the $USERS$ class. The values for coupling are $C(Disconnect, AssignResource) = 0.250$ and $C(Disconnect, AddPermission) = 0.267$, so a swap of the methods would indeed increase the value of coupling in total a bit.
- The method $Disconnect$ could be either swapped with $ChangeIP$ or with the method $DeassignResource$. However, there is no variation of the swaps that does increase the values of coupling anymore.

The next step of Rule 16 is to try to delegate some of the methods to other classes and thus to increase the value of coupling on class level.

- $Disconnect$ could be either moved to the $ADDRESSES$ or $Unused$ class. Moving it to $ADDRESSES$ would mean that C decreases from 1.0 to 0.235 for the class named $ADDRESSES$ and that C increases from 0.267 to 1.0 for the class named $RESOURCES$. The difference is $\Delta = -0.032$. The delegation does not improve the situation as a whole. Moving $Disconnect$ to $Unused$ would decrease C from 1.0 to 0.125 for the $Unused$ class and increase it from 0.267 to 1.0 for the $RESOURCES$ class. The difference is $\Delta = -0.142$, and a delegation is of no use again.

- Delegating $AssignResource$ to the $USERS$ class as a alternative step would be possible, but is does not change the value for coupling (as $\Delta = \pm 0$).

So, in both cases, the delegation does not maximize the values of coupling in the mean. With no more swaps or delegations possible we are done. The resulting diagram is displayed in Fig. 7.2.

7.4.2 Discussion and Improvements

By following the rules 14 to 16 we end up in a transformation that maximizes the values for schema coupling in every class. The approach is based on the assumption that the structure of the specification and the structure of the resulting implementation are, at least along general lines, comparable. Though this might not be the case in all situations, there are empirical evidences that at least the measures are correlated [4]. So, the approach presents a heuristic that is worth to be applied. Also the problems of scalability are influenced positively.

With the introduction of a threshold value ρ we are additionally able to control how many methods in the system class are to be delegated. Thus, an increase in the number of schema operations is not so much a problem anymore. Two other aspects of scalability are not addressed in this paper, but they can also be dealt with easily:

- When the specification is big and when it contains a lot of given sets or free-types, then the number of classes is very high. One solution to this problem is to extend Rules 3 and 10 by a simple manual step: as the mapping strategy uses classes as types, the user should decide about which of the given sets or free-types to map to classes and which to ignore (treating them as basic types). This extension could keep the resulting UML diagram smaller and also help shifting the view onto the specification a bit (depending at the situation at hand).

- When there are a lot of associations, then it would be possible to introduce association classes (between the classes representing given sets or free-types) to the UML model. This increases the number of classes, but in several situations this eases the mapping process of operations. Schema operations quite often only modify one state identifier in the state space, and here an association class could easily be used to hold such (getter and setter) methods.

Complex specifications still lead to complex diagrams, but as the classes are constructed in such a way that their internal connectivity is at a maximum, the resulting diagrams are, at least from a measurement perspective, not the worst ones.

One final limitation of the approach remains: when the values for coupling are all the same or when there are a lot of empty class fragments, then the assignment (due to rule 16) is at random first (in our case this happened with the assignment of $DeassignResource$ to the $Unused$ class). So, it is up to the user to reformat the resulting UML diagram - a drawback developers have to deal with anyway in situations when there are several class-candidates.

7.5 Conclusion and Outlook

This paper presents a set of rules for transforming formal Z specifications to UML in order to open the documents to a wider range of stakeholders. Existing approaches produce very useful UML diagrams, but the assignment of operations to classes still follows a simple heuristic, namely that of looking how often the operations refer to specific state-identifiers. As an extension to this strategy this contribution recommends to make use of slice-based measures. It introduces a coupling-based measurement method for the related schema operations and suggests to map the operations to classes in such a way that the average values for Inter-Schema Coupling stay at a, on the class level, high value.

The approach enables a wider range of analysis techniques and extensions that will be looked at in the future (and that will eventually be integrated into the ViZ environment[6] in one of the next releases). At first, the technique can be combined with existing techniques (like that of Idani in [15]) and thus help in producing additional variations of the resulting UML diagrams. Secondly, the generated UML diagrams can be assessed, either by looking at code smells (like large or lazy classes), by looking at exceptionally high values of coupling between some of the classes, or by combining the approach with a high-level treatment of object-oriented analysis and design. The results can then be used to reflect on the original specification itself.

There are still limitations that should not be concealed. As with other approaches the issue of inherent complexity is hard to solve automatically. Some improvements are possible, e.g. by different views onto parts of the specification or by making use of association classes whenever possible. The transformation rules do not necessarily lead to UML representations as generated by experienced developers by hand, but the approach provides a good picture of *what* is *in* the specifications and it puts the rules for the generation on a measurement-based solid ground. With that, it enables external validation steps and it supports the comprehension process of the involved stakeholders.

REFERENCES

[1] Andreas Bollin. *Specification Comprehension – Reducing the Complexity of Specifications*. PhD thesis, University of Klagenfurt, 2004.

[2] Andreas Bollin. Crossing the Borderline - from Formal to Semi-Formal Specifications. In *Software Engineering Techniques: Design for Quality*, pages 73–84. Springer, 2006.

[3] Andreas Bollin. Concept Location in Formal Specifications. *Journal of Software Maintenance and Evolution – Research and Practice*, 20(2):77–105, 2008.

[6] A description of the environment and the software can be found at http://viz.uni-klu.ac.at. Page last visited: September 2011.

[4] Andreas Bollin. Slice-based Formal Specification Measures - Mapping Coupling and Cohesion Measures to Formal Z. In Cèsar Muñoz, editor, *Proceedings of the Second NASA Formal Methods Sysmposium*. NASA Center for AeroSpace Information (CASI), April 2010.

[5] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

[6] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[7] Jeremy Dick and Jerome Loubersac. A visual approach to VDM: Entity-structure diagrams. Technical Report DE/DRPA/91001, Bull, 68, Route de Versailles, 78430 Louveciennes (France), 1991.

[8] Jermey Dick. Formalising the Informal: Linking Formal Methods to Informal Requirements. Inited Talk at FMICS'2004 as a co-located workshop of ASE 2004, Johannes Kepler Universitt Linz, Austria, September 2004.

[9] Sophie Dupuy, Yves Ledru, and Monique Chabre-Peccoud. An overview of RoZ: A tool for integrating UML and Z specifications. In *Proceedings of CAiSE'00*, pages 417–430, 2000.

[10] Houda Fekih, Leila Jemni, and Stephan Merz. Transformation des spècifications B en des diagrammes UML. In *Proceedings of Approches Formelles dans l'Assistance au Développement de Logiciels, $AFADL'04$*, pages 131–148, 2004.

[11] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[12] Mark Harman, Margaret Okulawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of coupling. In *Proceedings of the IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution. Boston, Massachusetts*, pages 28–32, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[13] Xudong He. PZ Nets - a formal method integrating Petri Nets with Z. *Information and Software Technology*, 43(1):1–18, 2001.

[14] Mike Hinchey, Michael Jackson, Patrick Cousot, Byron Cook, Jonathan P. Bowen, and Tiziana Margaria. Software engineering and formal methods. *Communications of the ACM*, 51(9):54–59, September 2008.

[15] Akram Idani. UML Models Engineering from Static and Dynamic Aspects of Formal Specifications. In Terry A. Halpin, John Krogstie, Selmin Nurcan, Erik Proper, Rainer Schmidt, Pnina Soffer, and Roland Ukor, editors, *Enterprise, Business-Process and Information Systems Modeling, 10th International Workshop, BPMDS 2009, and 14th International Conference, EMMSAD 2009, held at CAiSE 2009, Amsterdam, The Netherlands, June 8-9, 2009. Proceedings*, volume 29 of *Lecture Notes in Business Information Processing*, pages 237–250, 2009.

[16] Akram Idani and Yves Ledru. Object oriented concepts identification from formal B specifications. In *Formal Methods in Industrial Critical Applications, FMICS'04*, 2004.

[17] Akram Idani, Yves Ledru, and Didier Bert. Derivation of UML class diagrams as static views of formal B developments. In *7th International Conference on Formal Engineering Methods, ICFEM 2005*, pages 37–51, 2005.

[18] Stuart Kent. Constraint diagrams: Visualising invariants in object-oriented models. In *In Proceedings of OOPSLA'97*. ACM Press, 1997.

[19] Soon-Kyeong Kim and David Carrington. Visualization of formal specifications. In *In Proceedings Sixth Asia Pacific Software Engineering Conference (ASPEC'99)*, pages 102–109. IEEE Computer. Society Press, Los Alamitos, CA, USA, 1999.

[20] Soon-Kyeong Kim and David Carrington. A formal mapping between UML models and Object-Z specifications. *Lecture Notes in Computer Science*, 1878:2–21, 2000.

[21] Joachim Lessacher. Visualisierung von Spezifikationen – Transformation von formalen Z-Spezifikationen in UML-Diagramme (in German). Master's thesis, Alpen-Adria Universität Klagenfurt, 2007.

[22] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons Ltd., 1990.

[23] Timothy M. Meyers and David Binkley. An Empirical Study of Slice-Based Cohesion and Coupling Metrics. *ACM Transactions on Software Engineering and Methodology*, 17(1):2:1–2:27, December 2007.

[24] Roland T. Mittermeir and Andreas Bollin. Demand-driven specification partitioning. In *Proceedings of the 5th Joint Modular Languages Conference, JMLC'03*, Ausgust 2003.

[25] Tomohiro Oda and Keijiri Araki. Specification slicing in a formal methods software development. In 17^{th} *Annual International Computer Software and Applications Conference*, IEEE Computer Socienty Press, pages 313–319, November 1993.

[26] Linda M. Ott and Jeffrey J. Thuss. Slice based metrics for estimating cohesion. In *In Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81, Los Alamitos, CA, USA, 1993. IEEE Computer Society.

[27] D. Roe, K. Broda, and A. Russo. Mapping UML models incorporating OCL constraints into Object-Z. Technical Report ISBN/ISSN: 1469-4174, Imperial College of Science, Technology and Medicine, Department of Computing, 2003.

[28] Philip E. Ross. The exterminators. *IEEE Spectrum*, pages 36–41, September 2005.

[29] J.M. Spivey. *The Z Notation*. C.A.R. Hoare Series. Prentice Hall, 1989.

[30] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.

Appendix - Access Control Specification

The Access Control Specification is a sample specification (after Kim in [19]) for demonstrating the transformation process. Please note that the specification is not meant to be complete (e.g. an operation for creating resources would be needed, also the initialization could be handled in a different manner).

$$[USERS, RESOURCES, ADDRESSES]$$

AccessControl

$$\begin{aligned} &Permitted : USERS \leftrightarrow RESOURCES \\ &Assigned : USERS \nrightarrow RESOURCES \\ &IpAddress : ADDRESSES \nrightarrow RESOURCES \\ &Unused : \mathbb{P}\, RESOURCES \\ \hline &Assigned \subseteq Permitted \\ &Unused \cap (\text{ran}\, Assigned) = \varnothing \end{aligned}$$

InitAccessControl

$$\begin{aligned} &AccessControl \\ \hline &Permitted = \varnothing \\ &Assigned = \varnothing \\ &IpAddress = \varnothing \\ &Unused = \varnothing \end{aligned}$$

AddPermission

$$\Delta AccessControl$$
$$user? : USERS$$
$$resource? : RESOURCES$$

$$(user? \mapsto resource?) \notin Permitted$$
$$Permitted' = Permitted \cup \{user? \mapsto resource?\}$$
$$Assigned' = Assigned$$
$$IpAddress' = IpAddress$$
$$Unused' = Unused$$

ListUsers

$$\Xi AccessControl$$
$$resource? : RESOURCES$$
$$st! : \mathbb{P}\ USERS$$

$$st! = \text{dom}(Permitted \rhd \{resource?\})$$

AssignResource

$$\Delta AccessControl$$
$$user? : USERS$$
$$resource? : RESOURCES$$

$$(user? \mapsto resource?) \in Permitted$$
$$user? \notin \text{dom}\ Assigned$$
$$Assigned' = Assigned \cup \{user? \mapsto resource?\}$$
$$Permitted' = Permitted$$
$$IpAddress' = IpAddress$$
$$Unused' = Unused$$

ChangeIP

$$\Delta AccessControl$$
$$ip? : ADDRESSES$$
$$resource? : RESOURCES$$

$$ip? \notin \text{dom}\ IpAddress$$
$$resource? \in \text{ran}\ IpAddress$$
$$resource? \in \text{ran}\ Permitted$$
$$IpAddress' =$$
$$\quad \{i : ADDRESSES;\ r : RESOURCES \mid$$
$$\quad (i, r) \in IpAddress \land r \neq resource?\}$$
$$\quad \cup \{ip? \mapsto resource?\}$$
$$Permitted' = Permitted$$
$$Assigned' = Assigned$$
$$Unused' = Unused$$

DeassignResource

$\Delta AccessControl$
$user? : USERS$
$resource? : RESOURCES$

$resource? \notin Unused$
$(user?, resource?) \in Assigned$
$Assigned' = Assigned \setminus \{user? \mapsto resource?\}$
$Unused' = Unused \cup \{resource?\}$
$Permitted' = Permitted$
$IpAddress' = IpAddress$

Disconnect

$\Delta AccessControl$
$addr? : ADDRESSES$
$resource? : RESOURCES$

$resource? \in Unused$
$(addr?, resource?) \in IpAddress$
$IpAddress' = IpAddress \setminus \{addr? \mapsto resource?\}$
$Permitted' = Permitted$
$Assigned' = Assigned$
$Unused' = Unused$

PART IV

CONCEPT LOCATION AND MANAGEMENT

CHAPTER 8

CONCEPT LOCATION IN FORMAL SPECIFICATIONS

A. Bollin

Journal of Software Maintenance and Evolution - Research and Practice, 20(2):77-105, 2008.

Abstract

When kept up-to-date, formal specifications can act as valid artifacts for maintenance tasks. However, their linguistic density and size impede comprehension, reuse, and change activities. Techniques like specification slicing and chunking help in reducing the number of relevant lines of text to be considered, but they expect the point of change to be known a-priori.

This contribution presents a process model for concept location within formal Z specifications. It also considers those situations when the location is not even roughly known. The identification is comparable to the identification of regions with high cohesion. The approach is based on the idea of first transforming the specification to an augmented graph and, secondly, on the generation of spacial clusters.

8.1 Introduction

The use of formal methods helps in raising the overall quality of the software and the related software development process. Despite several myths and problems [6, 13] formal specifications have been used successfully more than once – they were and are used as test-

driver generators and form the basis for verification and validation steps [20, 22]. What is often ignored, though, is the fact that formal specifications provide valid resources during development and maintenance phases [4, 16].

One of the benefits of formal specifications is their density in writing down thoughts as they allow the authors to express requirements in a keen and compact way. However, this property – in combination with the raising size of today's systems – is one of the reasons for considering formal specifications to be too complex. Tools and approaches are needed to deal with complexity more than ever.

Formal specifications are complex "entities". Even simple textbook-specifications (of about 20 to 200 lines of specification text) typically contain dozens of dependencies between the related specification elements. Within the scope of this work these elements are called *prime objects* and they are, like program statements, the basic construction elements [16].

In order to guarantee the benefits of formal specifications (during evolutionary steps taking place throughout the life-cycle) it gets necessary to evolve and to *change* the specification, too. This change requires to first comprehend the specification. This can be demanding, and it gets really difficult when the maintenance personnel is not the author of the specification.

There are several ways in dealing with the compactness of specifications. When the location of the specification text to be changed is already known, smaller parts of the specification can be detached by slicing or chunking [17, 9, 2, 27]. Specification slices and chunks are well defined types of abstraction, and for Z-specifications [21] they do reduce the size of the text by 20% up to 60% [3].

However, typically, a change request might not contain a reference to the lines of specification text to be altered. Requests are often formulated in terms of changing/adding/deleting features or concepts [19]. For that reason concept and feature location plays an increasing role in software maintenance, and it will also play an important role during specification maintenance activities.

This contribution introduces an approach facilitating the comprehension process of complex formal specifications. Research dealing with specification slices and chunks so far assumes that either the point of change is known a priori, or that this point is to be found in an intuitive way (by linearly scanning through the specification text). The main contribution of this work consists of two closely related parts: firstly, a process model for specification comprehension, and secondly, fit into it, an approach for specification concept identification aiming at exactly those cases when not even a surmise can be made about the concepts' locations.

The contribution is structured as follows: Section 2 summarizes the idea behind concept location and defines the term "formal specification concept". Section 3 introduces a process model for concept location within formal specifications. It also introduces the notion of a specification cluster, a subset of specification elements which will be treated as a concept candidate. Section 4 presents a clustering algorithm for Z specifications. Section 5 demonstrates the applicability of the approach and addresses the issue of scalability. Section 6 concludes the work and provides hints to further variations of the approach.

8.2 Concepts and Concept Location

The Webster's dictionary defines the notion of a *concept* as an *abstract or generic idea generalized from particular instances*. In that sense it is a logical entity conceived in the mind. When talking about concepts in software engineering, the term *feature* is used very often. A feature then represents a selectable concept, which means that it is up to the user (and thus up to specific control sequences) whether the feature is executed or not. According to [15] *concept location* typically denotes a process that maps these domain concepts (or patterns) to code positions. The process of identifying patterns or abstract concepts "behind" the code is sometimes also called *plan recognition*.

By following the constructivist's views, existing approaches [14, 24, 18, 19] do build hierarchies of concepts that are related to parts of a unified representation of the program (in most cases the AST). By aggregating base concepts, higher level concepts are defined and stored in a library. When searching for concepts, AI techniques and fuzzy reasoning can be applied [8]. Code positions are typically part of several concepts, and to speed up the combinatorial problem when searching for concepts, special heuristics or constraint-satisfaction problem solving algorithms can be used [26].

For the purpose of this paper the notion of a *concept* also follows the constructivists' views. Concepts do get unique concept names and are, according to a taxonomy, organized in hierarchies of primitive concepts C (for example, $C_{Bubble-Sort}$ and $C_{Quicksort}$ belong to $C_{Sorting-Algorithm}$, and $C_{Sorting-Algorithm}$ belongs to $C_{Algorithm}$). Individuals are instances of a concept C. By making use of the description logic defined in [23, p.19] the following general definition of a concept can be provided (the domain, one is reasoning about, is expressed by the symbol Δ):

Definition 24 *A concept C is an intensional description of a class of individuals. It consists of a set of primitive concepts p and the transitive closure of their antecessor concepts. C^I, the interpretation I of the concept C, consists of the domain Δ^I and p^I, the interpretation of the primitives $p \in C$.*

According to Kozaczynski, Ning, and Engberts [15, p.1068], concepts "are not tightly related by syntactic structures, they are more closely connected by semantic relationships such as control flow, data flow, and calling relations". These relationships, sometimes also called *constraints*, facilitate the process of concept location.

8.2.1 Identification of Concepts

The technique of concept location is, in most cases, an intuitive process. Experienced users hardly can explain how concepts are identified; they navigate through the code and quickly separate relevant parts from irrelevant ones. When experience and familiarity with the code is not sufficient, more "structured" approaches are needed. Following [19], these approaches can be classified as follows:

a) *Pattern Matching*. Here, string-pattern matching algorithms are used (e.g. "grep"–ing for relevant keywords/identifiers) in order to identify concepts of pre-assumed names.

b) *Dynamic Analysis*. They are based on looking at the dynamic behavior of the code. The program is executed, and the program trace is analyzed. Here, a popular method is called software reconnaissance [25]. The code is annotated and executed twice, the

first time without the feature and the second time with the feature executed. Those parts of the code that are only executed in the second run are good candidates for starting the search for the assumed concept.

c) *Static Analysis*. Here, typically control and/or data flow is considered. By starting at the main entrance point of the program, tracing through the code, and by following explicit and implicit dependencies, it is assumed to speed up the identification of the exact location of the concept.

When taking a closer look at the above mentioned approaches, it turns out that not all of them can be applied to formal specifications:

ad a) String pattern matching works well for concept location within formal specifications. However, there are drawbacks. Without hints to suitable homonyms and synonyms this technique fails. In addition to that specifications are very compact. This compactness is achieved by making use of mathematical symbols, abbreviations and powerful operations. Developers of formal specifications typically do not label them with long and self-explanatory names, which also impedes looking for string patterns.

ad b) Dynamic analysis requires execution of the code. In general, specifications are not executable. Due to their declarative nature (as there is no explicit ordering of statements), a paper-and-pencil run is also not possible. Dynamic analysis is impeded.

ad c) The static analysis approach is mainly based on the identification of two dependencies that are not predominant in declarative systems: control and data-flow dependencies. However, as will be explained in Sec. 8.4.2, these dependencies can be re-constructed by the calculation of pre- and post-conditions within operations, and thus they can then be used to generate different types of specification abstractions [16].

By making use of pattern matching and slicing or chunking, the identified parts very likely represent concepts or features. But, there are two challenges concerning these static techniques:

- The number of dependencies within formal specifications is typically overwhelming. There is no heuristic telling the peruser which of the dependencies to follow.
- Slicing and chunking is based on a slicing/chunking criterion, thus on a specific position in the specification text. In most cases this position is not known a-priori.

As will be explained in Sec. 8.4.2, the static information yielded by a formal specifications is sufficient to solve these problems.

In most cases it is easier to state that some piece of text represents a concept than reasoning the other way round. With formal specifications this means that a specification slice might represent a concept, but it is unreasonable to state that every concept can be covered by a specific slice. Thus for the scope of this contribution, the approach in fact leads to *concept candidates*. The final validation of these candidates is, as is the same with program concept location, up to the user.

8.2.2 Concepts in Formal Specifications

First, let us start with a definition of a specification concept that borrows from the very general definition of a concept at the beginning of Sec. 2.

Definition 25 *A* **formal specification concept** *is a coherent, abstract (or generic) pattern of specification text that is generalized from particular instances of the specification. It can be understood and recognized as a whole even when standing alone.*

The definition is closely related to those of a Burnstein Chunk [7] or a specification cliché [2, p.50]. The first part of the definition deals with its representational form, the second part ensures that a specification concept has a meaning that is unambiguously understood within its context.

With respect to Def. 24 it should be clear that the interpretation of a specification concept strongly depends on the interpretation of its related parts. These parts or basic objects are called primes of a specification.

Definition 26 *A* **specification prime** *(also called prime object) represents the basic entity of a specification. It is built out of literals of the specification and forms logical, syntactic, or semantic units.*

In that sense a specification prime constitutes those primitives that form the smallest, basic concepts expressed in a formal specification. It is a *syntactically coherent sequence of literals within a state-based specification, forming semantic entities that can be paraphrased by a short sentence in natural language*. An example would be a Z predicate checking whether some names are in a specific set (called $known$) or not: "$name? \notin known$".

When primes are aggregated, they do form so-called *higher-level primes*. An example for a higher-level prime is the schema expression "$[\Xi Elevator \mid Request \cup UpCalls \cup DownCalls = \varnothing]$". It consists of two basic primes, $\Xi Elevator$ and a predicate describing properties of some sets. A more detailed introduction to Z primes can be found in [2, p.43].

With such basic elements a more precise definition of a specification concept can be provided:

Definition 27 *A* **formal specification concept** C_{FS} *within a formal specification* Ψ *is a concept according to Def. 24, where the domain* Δ *is the domain of the formal specification* Ψ*, and the primitive concepts* p *are represented by prime objects.*

Definition 28 *The* **interpretation** *of the formal specification concept* C_{FS} *is determined by aggregating the primes' interpretations according to Def. 24 (*$C_{FS}^I == (\Delta^I, \{p : Prime \mid p \in C_{FS} \bullet p^I\})$*).*

In order to be able to reject primes as concept candidates, the following definition is yielded:

Definition 29 *Every specification prime* $p_i : \Psi$ *is element of* C_{FS} *when at least one of the following conditions holds:*
(i) There are (syntactic and/or semantic) dependencies between prime p_i *and primes already in* C_{FS}*.*
(ii) The absence of prime p_i*'s semantics in* C_{FS} *impedes the recognition of the concept in mind.*

Thus the semantics of the concept or concept candidate is constructed by the subsumption of the semantic interpretations of its related components.

8.2.3 Concept Location within Formal Specifications

Formal specification concept location is still cumbersome. A program-like static technique for searching for concepts is impeded by the huge number of dependencies and the often missing "starting point". However, this hurdle can also be seen as an advantage. As noted in [15], concepts often express themselves by their constrains, their semantic relationships. And looking for specific groups of constraints can be compared to clustering the specification.

Definition 30 *A* cluster *is a group of the same or similar objects gathered or occurring closely together.*

In our case the objects are primes. Similarity and closeness might be expressed by several properties, for example similar type or number of dependencies the objects depend upon. For the scope of this paper the reachability via dependencies will be used as an argument for closeness.

Definition 31 *A* **specification cluster** *is a group of primes. A prime p is element of the cluster when the number of (direct or indirect) dependencies to elements already in the cluster is higher than the number of dependencies to elements outside the cluster.*

Generally spoken, the clusters we are looking for are sets of primes with high cohesion in respect to direct and indirect relationships.

Specification clusters (as well as slices and chunks) are being built by the aggregation of prime objects and by regarding specific types of dependencies in between them. This property coincides with the definition of a specification concept. This is the main reason for stating that *specification clusters (as well as slices and chunks) are to be treated as concept candidates*.

The remaining arguments (reinforcing that concept location can be supported by looking for specification clusters) are as follows:

- Several dependencies can be considered once at a time as it is not always sensible to follow only a single dependency.
- By clustering, related objects are brought together, and dissimilar objects are separated. No artificial ordering of prime objects is introduced.
- Clusters do not necessarily have strict boundaries. The same holds for concepts and their related primes - they can be part of other concepts, too.

There are several clustering algorithms, each one with its own advantages and disadvantages (for an introduction see [12]). Cluster algorithms depend on at least two inputs: a pattern vector and the number of clusters to be identified. In our case this allows for adjusting the granularity of the search space – something that will be useful when narrowing down the search space. The clustering approach presented in this contribution is based on clustering spatial properties of a directed graph. The approach is embedded into a process model for specification concept location, and the next section introduces the underlying model in more detail.

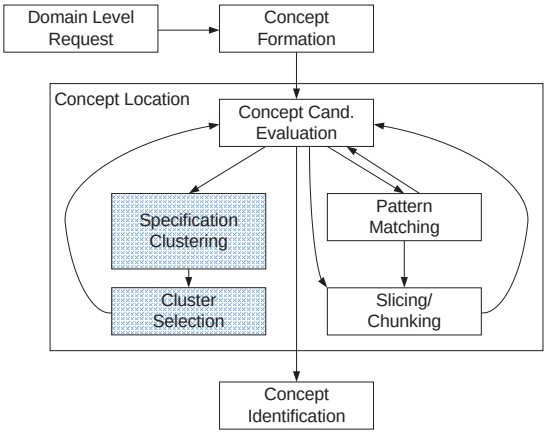


Figure 8.1 Concept location as an iterative process. Concept candidates are identified and evaluated in respect to their suitability. Either clustering techniques or patter-matching and slicing/chunking are applied. (The approach described in details in Sec. 4.3 is marked grey, slicing/chunking is described thoroughly in [2].)

8.3 Concept Location Process Model

The search for relevant pieces of specification text is never purely top-down or bottom-up, it is a mixture of both – and consists of iterative steps. In addition to that and being confronted with an unknown specification, one has to start with an initial concept in mind, a concept that is validated against concept candidates. Figure 8.1 presents a model, which consists of four main activities (and which is a refinement of the model already presented in [10]).

- *Domain Level Request*. The feature or concept to be located in the specification text is formulated, and typically comes from "outside".
- *Concept Formation*. The peruser has to form a mental image of the problem, thus it requires to comprehend the request from the outside world. With this, the process of concept location is about to start.
- *Concept Location*. It is an iterative process consisting of several sub-activities:
  - *Concept Candidate Evaluation*. With the problem in mind the peruser takes the concept candidate (at the beginning the intuitively most suitable specification fragment) and checks whether it matches to the concept in mind or not. Then he or she decides between accepting the candidate or stepping (again) into the concept location process.
  - *Specification Clustering*. The candidate either contains no parts of interest or it is still too large. Clustering is applied in order to narrow down (or broaden) the search space.
  - *Cluster Selection*. Clustering yields a set of clusters. Each of these clusters has different semantics the peruser has to understand. One or more suitable clusters have to be selected as potential candidates, others dismissed.

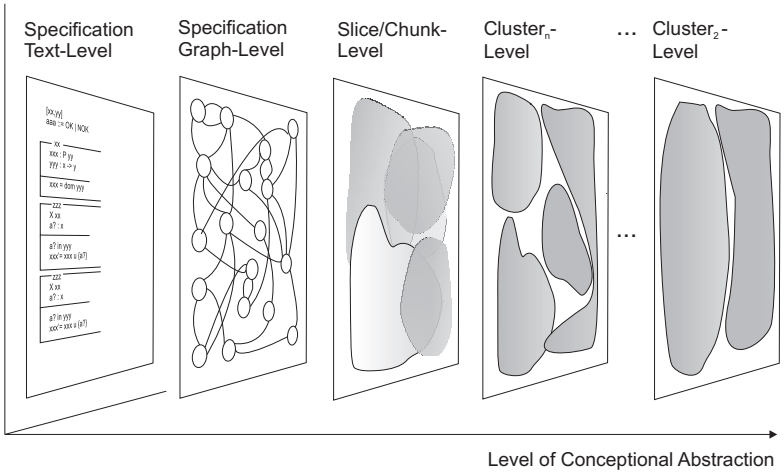


Figure 8.2 Formal Specifications can be transformed to a graph mapping primes to vertices. These graph forms the basis for dependency analysis. It is possible to carve out slices, chunks, or clusters by aggregating vertices.

  - *Pattern Matching*. When the search space is large, pattern matching strategies can be applied. The peruser browses or "greps" through the candidate(s) and looks for relevant labels or identifiers. This might help separating suitable from less suitable parts of the document.
  - *Slicing/Chunking*. When possibly relevant locations in the specification text are identified, slicing and chunking can be applied. This helps in modifying the search space again, but also guarantees (when necessary) syntactical completeness and correctness.

- *Concept Identification*. The location process results in a set of candidates that very likely do contain those parts (primes) to be changed. These parts have to be marked (or labelled) as being part of the change process.

The steps of activities are introduced in a top-down manner, but it is also possible to start bottom-up. The peruser then has to decide whether he or she is satisfied with the candidate or not. Clustering and slicing/chunking techniques can, as a next step, be used to enlarge or narrow down the set of relevant concept candidates. In any case one is following the iterative steps of the model.

As the clustering approach is not to be mixed up with the strategy of "divide and conquer", it needs more explanation. The data-set is not divided into several separate parts that are not overlapping. In the above process model a partitioning takes place, too (which is the reason for speeding up the search), but the partitions might change and might contain parts of candidates that were excluded one loop before.

The basis for the specification comprehension process is the specification itself (Specification Text- or Graph-level in Fig. 8.2). The representation can be either seen as (linear) specification text, or as a graph containing all relevant information. According to [16] a graph is more likely the "true" representation of a specification, as there is no linear ordering. However, as will be explained in Sec. 8.4.1, both representations can be transformed

loss-lessly to each other. When starting to look for abstractions with well known semantics, slices and chunks (Slice/Chunk-Level in Fig. 8.2) are considered. The bad news is that slices and chunks are typically quite large. Experiments indicate that a Z-chunk tends to contain 50 prime objects, and slices are often of the same size as the original specification [4]. Additionally, matching the general description of a concept of the world to a slicing criterion (the exact location of at least one prime object) is quite difficult.

Clusters add an orthogonal view onto this problem (Cluster-levels in Fig. 8.2). The peruser decides about the granularity and might start with n clusters, which also have a semantic driven by the included prime objects. As we will see later there is no "optimal" number of clusters. However, it still is up to the user to decide whether some clusters match the target concept or not. He or she might start to increase the number of clusters and thus "sharpen" the concept candidates. On the other side he or she might reduce the number of clusters, increasing the level the abstraction. The disadvantage of providing the number of clusters at the beginning rapidly gets an advantage as with this input the level of abstraction can be "controlled" within some boundaries.

Finally, when identifying a suitable set of cluster candidates, the target prime objects are tied down. Then the level of abstraction can be decreased again, and, with the primes in mind, slices and chunks can be generated in order to identify those parts that are affected by a potential change.

8.4 Clustering Algorithms

With these ideas in mind the approach has been evaluated on formal Z specifications to be found in textbooks and literature [9, 11]. For clustering (besides the number of clusters) the pattern vector is the most relevant input. It is based on properties of the graph, and the subsequent section introduces the necessary background. Sec. 8.5 finally addresses the issues of applicability and scalability of the approach.

8.4.1 Basic Transformation

At first the formal specification is transformed into a representation called *specification-relationship net* (SRN). The SRN is a directed graph that captures all primes present in the specification. It is defined independently of particular specification languages, however, the rule set to translate a specification in a syntax- and semantic-preserving manner to an SRN has to be defined in a language dependent manner. The SRN is then further augmented by *semantical structural information*, yielding an $Augmented\ SRN$ ($ASRN$). This information explicates dependencies between prime objects due to *type* and *variable declarations*, as well as definition and use *(def/use)-information*.

One major motivation for the generation of an $ASRN$ was to deal with a representation that expresses, on one hand, the absence of a specific ordering within specification elements itself and, on the other hand, the close relationships between specification primes. In fact, both forms of representation (text and ASRN) are interchangeable. The transformation basically maps primes to vertices in the graph. Concerning the structure of the net, the idea behind an SRN is that primes are enclosed by typed vertices (*Start* and *End* vertices), thus defining some sort of higher-level primes. Fig. 8.3 shows how the SRN is used to represent a schema of a Z specification. Z-constructs are translated to SRN-primitives. Primes that consist of other primes (such as Z-schemata) contain vertices that reference

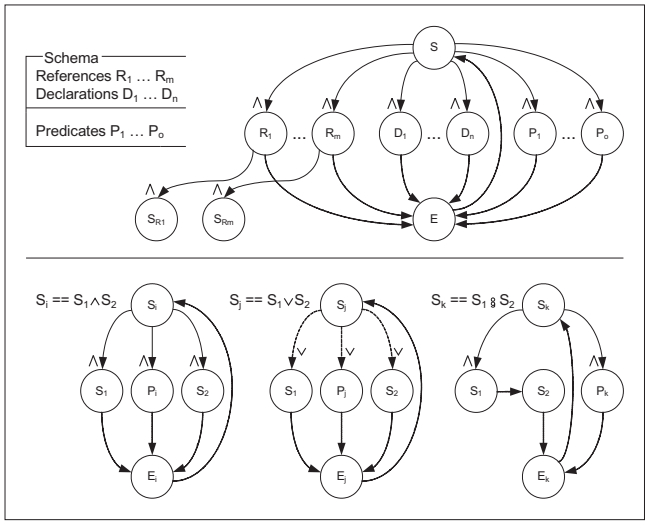


Figure 8.3 Mapping Z-specifications (at top left) to an SRN. The idea is to enclose the specifications' prime objects between structure preserving vertices. Logical combinations (at bottom) are resolved by typed arcs.

to those primes. Thus the general structure ensures that a hierarchy of primes and its enclosing structures are being built.

To summarize, an augmented SRN is an extension of an SRN, where vertices are getting attributes describing the "usage" of identifiers belonging to that vertices. A typical augmentation is demonstrated in Fig. 8.4. Besides the line numbers (L), it consists of the source text (as annotations A), the definitions of variables (D), the use of variables (U), type declarations (T), and input/output channel definitions (C).

8.4.2 Dependencies

As reachability in the $ASRN$ is put on a par with the notion of relatedness (which will be later an argument for cluster generation) this section explains when and under which conditions vertices in the graph are connected by arcs. In fact, in an $ASRN$ there are four classes of arcs, each of them represents different types of relationships.

8.4.2.1 Sequential Control Arcs Though the ordering of predicates within specifications is not relevant, pieces of text still rely on a logical or linear order that has to be preserved (e.g. the start of a schema box has to be before its predicates). This is done by making use of *sequential control arcs*. In order to deal with logical (boolean) operations, the sequential control arcs can be typed as either AND or OR control arcs. All arcs in Fig. 8.3 do belong to this class.

8.4.2.2 Declarational Arcs Identifiers typically have to be declared before they can be used. Depending on the specification language, there will always be dependencies that exist only due to the syntactic and semantic rules of the language (including the notion of scope). This does not refer to deep semantics of a specification, but it defines those dependencies that a good Z-checker would identify. From the parsers' perspectives, however,

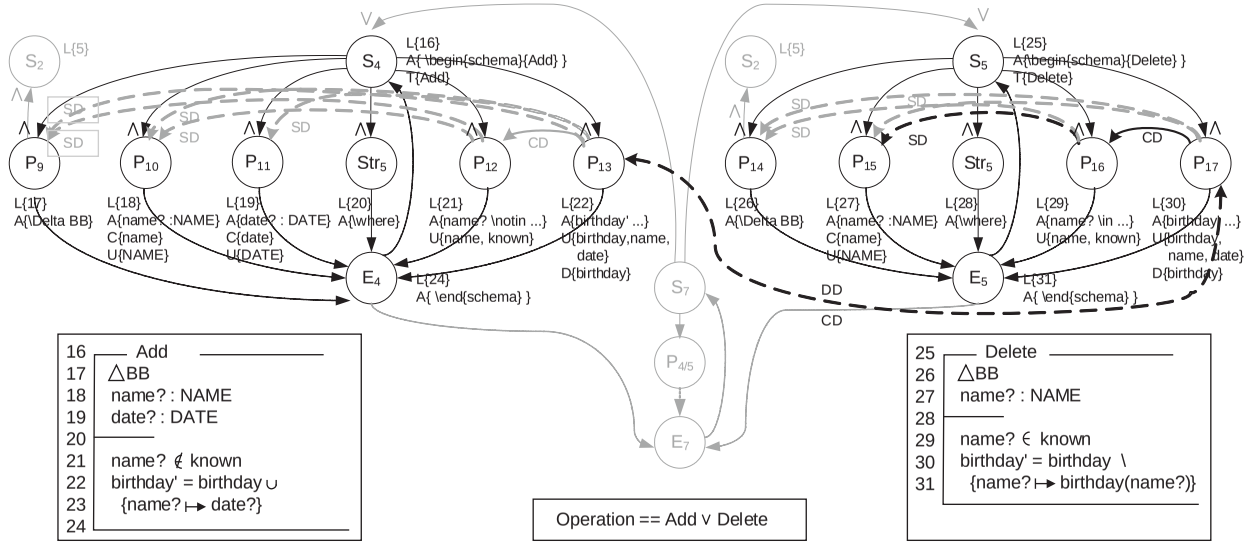


Figure 8.4 The *Add* and *Delete* operation schemata (bottom) of the birthday-book specification (see App. B) are mapped to an *ASRN* (top). Vertices do get annotations, and arcs are labelled according to the class they do belong. Dependencies are made explicit. (For reasons of readability irrelevant parts are greyed out).

these dependencies are beyond simple syntax. It is legitime to entitle this dependency as of being semantically structural. They are expressed by *(semantically) declarational arcs* in the $ASRN$. In Fig. 8.4 there are several arcs (labelled with SD) expressing this type of dependency. For example, prime P_{16} uses the identifier $name?$, which has to be declared first (to be seen at prime P_{15}).

8.4.2.3 Control Dependency Arcs The definition of *control dependency* can be kept rather simple when based on pre- and post-condition analysis between primes. By following the arguments in [9], in Z a prime q is control-dependent on a prime p when p potentially decides whether q is applied or not. This means that post-condition primes (after-state primes) are control dependent on pre-condition primes. The same idea holds when elements of the specification are logically combined. If, again in Z, a schema object S_1 is conjuncted to a schema object S_2, all post-conditions of S_1 are control dependent on the pre-condition of S_1 and S_2, and all post-conditions of S_2 are control dependent on the pre-condition of S_2 and S_1.

The identification is straight forward and, due to the structure of the graph, reduced to a reachability problem. One has to look for primes that do have defined (annotated by a D) identifiers (thus they are post-condition primes) and for primes that do not refer to any after-states. When they are reachable according to their scope then there is a control-dependency between them. In Fig. 8.4 control dependency is expressed by the arcs labelled with CD. E.g. P_{17} is control dependent upon P_{16}, as it is reachable from P_{16} via sequential control arcs and P_{16} does not define any identifier, indicating a pre-condition prime.

8.4.2.4 Data Dependency Arcs The identification of *data dependency* is quite similar to the identification of control dependency. A prime q is data-dependent on a prime p, when data potentially propagates from p to q through a series of state changes. Again the $ASRN$ eases the identification, as one has to look for identifiers at primes that are, within declarational boundaries, redefined at one prime (thus annotated by a D) and used at another prime (annotated by an U). Fig. 8.4 shows an $ASRN$ where the two operation schemata Add and $Delete$ are logically combined. Besides control dependencies, there is data dependency between primes P_{17} and P_{13}, as the identifier $birthday$ is re-defined (D) and used (U) at the opposite primes each.

With the structure of the $ASRN$ in mind one can also start to look for sets of primes that are closely (in terms of distance in the graph) related to each other. Relatedness then has to do with spatial reachability across different types of arcs, and the subsequent section explains how clustering techniques can be used to identify these regions.

8.4.3 Cluster Identification

Clustering is an approach that groups pattern vectors which do have similar characteristics in some respects. According to [12] the process of cluster identification is divided into several steps:

- *Feature selection*. Here, all attributes on which the clustering is to be performed have to be defined. As the relevant information (for the problem at hand) has to be encoded this is a critical step.
- *Cluster algorithm selection and clustering*. There are several types of clustering algorithms. Examples are partitioning clustering (e.g. $KMeans$), fuzzy clustering (e.g.

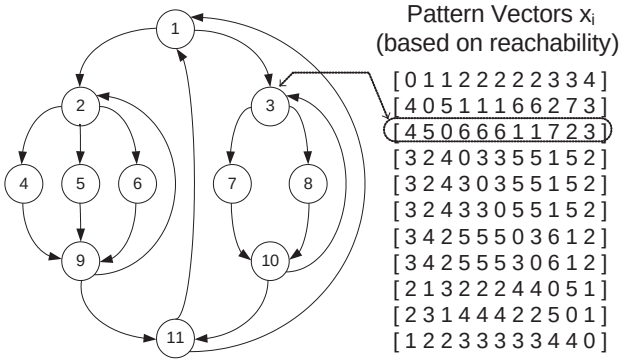


Figure 8.5 Pattern vectors for a simple SRN graph. The minimal length of the path between two vertices is taken as the attribute discriminating the relationship between the two vertices, stored in the pattern vectors x_i.

FCM), or hierarchical clustering. A proximity measure and a clustering criterion have to be selected.

- *Validation*. The resulting clusters are not known a-priori. So the final partitions have to be evaluated in respect to usability and appropriateness.
- *Interpretation*. The resulting clusters do have specific semantics that have to be interpreted correctly. One has to answer the important question: what does this cluster mean?

8.4.3.1 Feature Selection The first and very sensitive step is called feature selection. The arguments provided in Sec. 2 indicate that one is interested in grouping those parts of the specification which are closely correlated via dependencies. In other words, for the $ASRN$ it is reachability that counts. It makes sense to take prime objects (vertices in the $ASRN$) as data points and examine SRN-paths in the graph. For every prime the cost for reaching other primes in the graph can be calculated by identifying the length of the shortest path. For every vertex this yields a distance vector, and each vector discriminates the prime from the others. Fig. 8.5 presents a small graph (which is typical for structures to be found in $ASRN$s) and its related pattern vectors.

By searching for the shortest path, a simple cost function can be defined. The definition presumes that $minlength$ is a function determining the shortest path between two vertices in the $ASRN$.

Definition 32 *For all n vertices v, with v element of the $ASRN$, an n-dimensional pattern vector x_v ($v = 1 \,..\, n$) is defined. For every vertex x_v the set of paths P to vertices x_l ($l = 1 \,..\, n$) is calculated, and $x_v[l]$ is defined to be the minimum of the lengths of the paths in P ($x_v[l] = minlength(paths(x_v, x_l))$). The length is set to -1 when there is no path.*

For every vertex in the graph the vector represents the minimal costs in reaching all other vertices. As an example, in Fig. 8.5 the vertex number 3 gets the feature vector x_3 (line 3) attached. The vector tells us that vertex 1 is reachable from vertex 3 by the cost of 4 (over vertices 7, 10, and 11), and vertex 11 is reachable from vertex 3 by the cost of 3.

Centers c:

3.09	4.08	1.30	5.08	5.08	5.08	1.47	1.47	6.07	1.15	2.11
0.83	1.60	1.69	2.56	2.56	2.56	2.67	2.67	3.37	3.47	1.94
3.00	1.43	3.99	1.81	1.81	1.81	4.99	4.99	1.07	5.37	2.03

Membership Matrix U:

0.10	0.06	0.86	0.05	0.05	0.05	0.87	0.87	0.05	0.67	0.10
0.78	0.15	0.09	0.15	0.15	0.15	0.09	0.09	0.22	0.26	0.78
0.13	0.80	0.05	0.81	0.81	0.81	0.04	0.04	0.74	0.08	0.12

Maximum of U:

0.78	0.80	0.86	0.81	0.81	0.81	0.87	0.87	0.74	0.67	0.78

Figure 8.6 Results of the calculation of the pattern-vector's centers for the SRN (by applying $FCM(data, 3)$ to the data set) in Fig. 8.5.

8.4.3.2 Cluster algorithm and criterion The second step is cluster algorithm selection. As one is interested in separating the data in respect to the pattern vectors, we have to deal with partitioning algorithms. As it is very likely that primes can be part of different concepts (with different probabilities) at a time, the fuzzy c-means algorithm [1] has been chosen in order to express memberships of different extents.

For reasons of simplicity the proximity measure is the Euclidean distance. The clustering criterion is defined as to minimize the following objective function J_m:

$$J_m = \sum_{i=1}^{N} \sum_{j=1}^{C} u_{ij}^{m} \|x_i - c_j\| \tag{8.1}$$

where x_i represents the pattern vector defined above. m is a real number greater than one (in our case set to 2.0), u_{ij} is the degree of membership of pattern vector x_i in cluster j, c_j is the center of cluster j, N is the number of pattern vectors, and C is the number of cluster centers. Fuzzy clustering is an iterative optimization, where the values for u_{ij} and c_j are calculated.

Concerning the example in Fig. 8.5 one can apply the FCM algorithm with the argument of 3 clusters to be identified. Fig. 8.6 summarizes the results of the FCM algorithm. It yields the cluster centers and, for every vertex, a membership value u_{ij}. Those indices containing values equal to the maximum value of the rows n of matrix U are candidates for the cluster n. In our case this leads three clusters containing the following vertices:

- Cluster 1: V_3, V_7, V_8, V_{10}
- Cluster 2: V_1, V_{11}
- Cluster 3: V_2, V_4, V_5, V_6, V_9

This is exactly what was expected. In our case a user of the graph can easily observe that the vertices in the clusters are reachable by just a few arcs. However, with larger graphs this get much more difficult, and a(n) (semi-)automated approach is likely to be helpful.

8.4.3.3 Cluster validation The objective of clustering is to seek clusters where the related data vectors exhibit a high degree of membership. However, when there are n vertices, it is possible to apply $FCM(data, C)$ with argument C running from $2 \,..\, n$. The

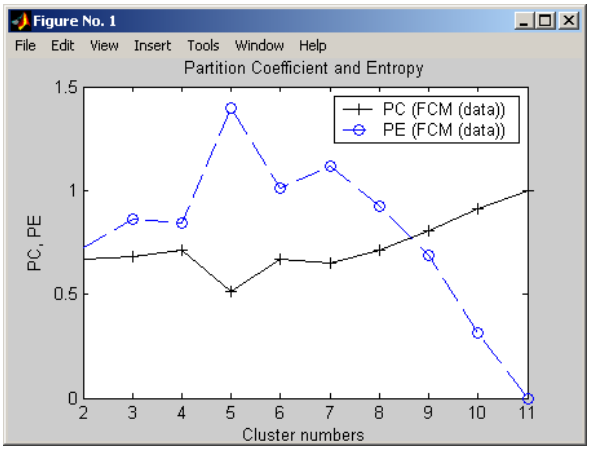


Figure 8.7 PC and PE are measures describing the fuzziness of the generated clusters. For the graph in Fig. 8.5 the values are calculated.

important question is then how "good" the generated clusters are. Here, the answer of the appropriateness of clusters cannot be given automatically, but there are clues indicating special properties of the cluster. These properties describe, e.g, how "fuzzy" a cluster is, or how "crisp".

The approach presented so far does not suggest a specific number of clusters to be initially generated. In fact, the question of the "ideal number of clusters" is interesting, but hard to predict. There are two possibilities to deal with this situation:

- Tell the user, that $2\,..\,C$ clusters are possible and let him or her decide freely about the cluster's generation.
- Calculate all possible sets of clusters and provide some hints about the clusters' internal properties. Then the user can decide about the cluster's generation.

The latter strategy seems to be more appropriate, as it still lets the decision up to the user and provides as much information as possible at that time. Typically, properties are described by so-called *validity indices*. Two helpful indices are the *partition coefficient* PC and the *partition entropy* PE. They are defined as follows [12, p.137]:

$$PC = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} u_{ij}^2 \tag{8.2}$$

$$PE = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} u_{ij} * ld(u_{ij}) \tag{8.3}$$

The PC index (8.2) tells about how fuzzy the cluster is. The closer the value to $1/C$, the crisper is the clustering. As can be seen in Fig. 8.7, the index is between 0.5 and 1. The higher the number of clusters, the sharper the result. Theoretically PC is a monotone increasing function when all clusters are fuzzy. In our case PC has a "sharp" bend at 5, indicating that there the fuzzyness decreases. It might be a good idea to start looking around that number of clusters first.

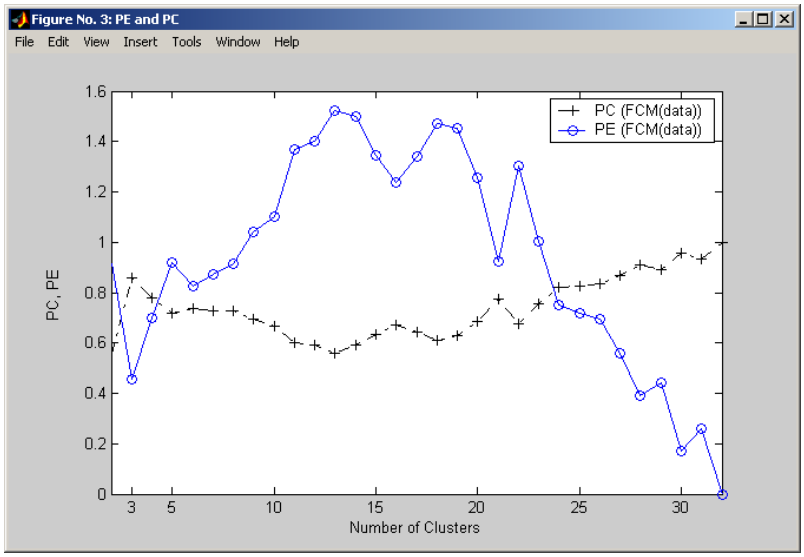


Figure 8.8 PC and PE values for 32 different clusters of the *Elevator* specification – with the focus on start-vertices.

The PE index (8.3) works the other way round: the nearer the values of PE to the upper bound $ld(C)$ the bigger is the absence of any clustering structure. Here a number of 2-4 clusters seems to be most interesting.

8.4.3.4 Cluster Interpretation Finally the generated clusters have to be interpreted. The pattern vectors (that are grouped into clusters) tell us about the similarity in costs of reaching neighboring vertices. In our case this means that vertices with similar distance vectors are closer to each other. The semantics of the cluster is then defined by (i) the semantics of the primes (thus the ASRN's vertices) and (ii) the type of dependencies in between them. So it is the same as with specification slices and chunks. Clusters do get their meaning in a neat way: by aggregating the meaning of the subgraph of the $ASRN$.

8.5 Evaluation

8.5.1 Case Study

The *Elevator* specification is a formal specification of a simple elevator control system to be found in textbooks and has been taken to discuss (quasi-) dynamic specification slicing in [9]. The task is to detect a bug as the elevator is sometimes moving into the wrong direction when pressing a button. So, all we know is that a *button* is pressed and that the *direction* of the *movement* is not as expected. The task of looking for the reason(s) of this malfunction is difficult, especially when one is not the author of the specification. In order to identify the location of the bug, one has to understand the concepts behind the specification. The concept location model presented in Sec. 8.3 is taken for their reconstruction. An annotated *Elevator* specification (including the hint to the bug) is provided in App. C.

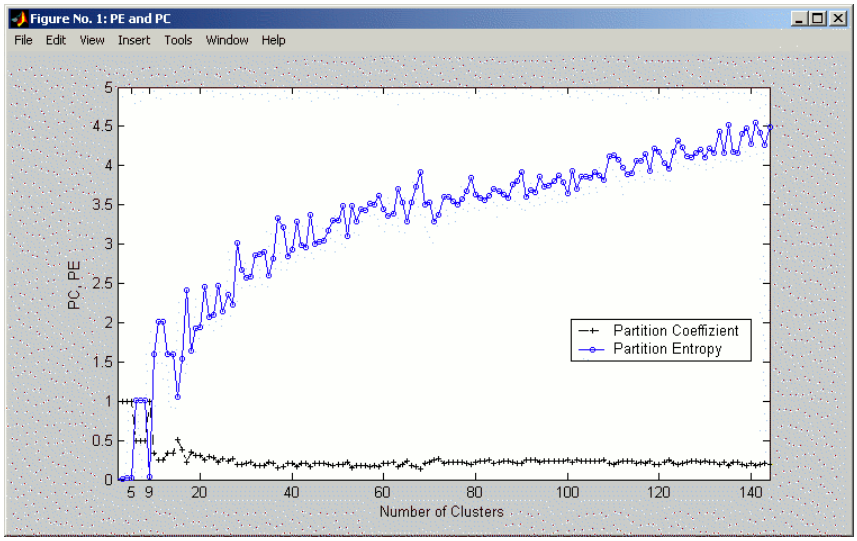


Figure 8.9 By looking at the *ASRN* of the *Elevator* specification 144 different clusters (with the focus on primes representing predicates) can be generated and their PC and PE values calculated.

Supported by a prototype tool (for details see [2, p.171]) the following activities are handled:

1. Understand the general concepts and the structure behind the specification.
2. Locate concept candidates and match them to the bugs description.
3. Narrow down the location of the concept.
4. Identify all relevant primes.
5. Change the relevant parts of the specification.

The process model does not help in avoiding to read the specifications' text, but it helps in narrowing down the search space quickly. Looking at the ASRN one realizes that the specification is quite complex. It contains 349 vertices, 1096 control, and 1212 data dependencies (direct and indirect). 144 vertices represent predicate primes, 32 vertices are start vertices.

At first, for reasons of orientation, the general structure behind the specification has to be understood. One decides to look at the syntactic structure of the specification and to do a simple clustering regarding the 32 start vertices (as they represent higher-level primes). The goal is to identify those regions that are (spatially) closely related. When taking a look at Fig. 8.8, one can see that the clusters are rather crisp when generating three of them (also see Fig. 8.11 in App. C). There are 3 closely related regions, and mapping the vertices back to the specification results in the following three *concept candidates* (containing the set of high-level primes) :

1. Cluster 1 contains the primes: $BasicMoveUp$, $BasicMoveDown$, $ChangeUpToDown$, $ChangeDownToUp$, $RestartMovingUp$, $RestartMovingDown$ and the primes related to their logical combination.

2. Cluster 2 contains the primes: $Elevator$, $InitElevator$, $ElevatorButtonEvent$, $Floor\ ButtonEvent$, the related given-set declarations, and their logical combination.
3. Cluster 3 contains the primes: $NoRequestOrCalls$, $OpenDoor$, $CloseDoor$, and their logical combination.

The first one deals with the movement of the elevator, the second one with the states, and the third one deals with opening and closing doors. As there is no problem with the doors, the primes in concept candidate 3 are excluded from the search in the future. This is one of the benefits of the definition of a specification concept: it is often easier to state that a set of primes is NOT within the concept to match.

No one of the candidates matches the original concept, thus we step further into the cycle and generate clusters regarding all 144 predicate primes and their dependencies to other primes. Fig. 8.9 shows the values for the crispness of the generated clusters and we notice, that 5 or 9 clusters seem to be interesting. We now can adjust the granularity. We generating 5 clusters and regarding those identifiers with data-dependency arcs:

- *Cluster 1*: Deals with door states (*Dir, Door*, contains 20 primes – excluded).
- *Cluster 2*: Deals with the elevator's movement (*UpCalls, DownCalls, CurrentFloor*, 26 primes).
- *Cluster 3*: Deals with the events (*Requests, CurrentFloor*, 11 primes).
- *Cluster 4*: Coordinates the idle-state (*CurrentFloor*, 23 primes).
- *Cluster 5*: Coordinates the doors and events (*CurrentFloor, Dir, Door*, 64 primes – excluded).

From this the identifiers *CurrentFloor, Dir, DownCalls,* and *UpCalls* of clusters 2 and 3 seem to be most relevant in respect to the concept of out problem in mind - however, still too many primes are to be regarded. We decide to produce clusters that are smaller ($n = 9$). This leads to nine clusters (and again focus on identifiers with data dependencies):

- *Cluster 1*: Moves the elevator down (*CurrentFloor, DownCalls*, 12 primes – excluded).
- *Cluster 2*: Operates on the door (*CurrentFloor, Dir, Door*, 64 primes – already excluded).
- *Cluster 3*: Operates on the door (*Door*, 9 primes – already excluded).
- *Cluster 4*: Starting to move (*UpCalls, DownCalls, CurrentFloor*, 14 primes – **candidate?**).
- *Cluster 5*: Idle state (*CurrentFloor*, 10 primes – excluded).
- *Cluster 6*: Idle state and initialization (*CurrentFloor*, 13 primes – excluded).
- *Cluster 7*: Request handling (*Requests, CurrentFloor*, 11 primes – **candidate?**).
- *Cluster 8*: Operates on the door (*CurrentFloor, Dir, Door*, 64 primes – already excluded).

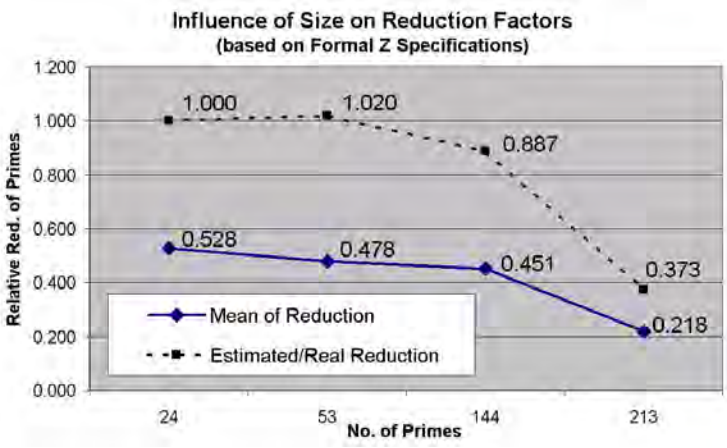


Figure 8.10 The effect of reduction increases with the size of the specification. The bigger the specification the lower the $Mean\ of\ Reduction = Primes_{after}/Primes_{before}$. When the reduction increases at the same ratio than the specification, then the values for the estimated reduction would be 1. Values less then 1 indicate better performance.

- *Cluster 9*: Changing the direction (*Dir*, 11 primes – excluded).

We evaluate the clusters: when the elevator is already moving then there are no problems. Using pattern matching and the findings of above, clusters 1 up to 3, and 5, 6, 8, and 9 might be excluded from the search. So we suppose that the problem is located within cluster 4 or 7. Cluster 4 contains 14 primes and we take a closer look at it - thus reading the related specification text.

Finally we identify one prime (in Fig. 8.11 it is the vertex entitled "343: L(55/39) FloorButtonEvent" at the left) that sets the direction of the movement within the floor button event operation schema. With that we identified potential primitives of a concept candidate. There is still no guarantee that the candidate matches the problem description, and thus we decide to generate a Burnstein chunk on this prime element. It yields four schemata: *Elevator, InitElevator, FloorButtonEvent, and CloseDoor*, and thus reduces the size of the specification drastically.

By skimming through the remaining specification we find out that one predicate in $FloorButtonEvent$ has the bug inside. Due to a copy-paste error the reaction to an event was wrong. Instead of handling $upCalls$ correctly, $upCalls$ were treated as $downCalls$.

The process of concept location aims at speeding up the identification of relevant parts of the specification. However, there is the chance of digging into parts that are not relevant, too. Clusters provide some feedback and can give some hints about their semantic interpretation. But still it is up to the user to match the concept carved out of the specification text to the concept in mind.

8.5.2 Reflection

The usefulness of the suggested model depends, at least, on the following two aspects: firstly, the scalability of the approach, and secondly, the scalability of the algorithms behind. Really big formal specifications are rare, but the approach has been tested with

several specifications of growing sizes. Among the largest is the specification of the ICT window manager of the "Andrew distributed system" developed at the Carnegie-Mellon University. Its ASRN contains more than 500 primes (213 are higher-level primes) and 2633 dependencies. The specification is described in [5, p.183ff].

The question of scalability is directly related to the existence of conceptual structures behind the underlying specifications. Clustering, slicing, and chunking algorithms are based on the assumption that there are interconnected regions within the specification. Thus the approach scales when (a) these regions exist, and (b) when, with growing sizes of the specification, these regions stay at manageable size.

ad (a). In [3] it has already been shown that chunks typically consist of 50 primes in the mean. Though it is possible to construct specifications that do contain regions of more than several dozens of interwoven primes, none of the case studies analyzed in [3] and looked at so far contained even complexer constructs. Developers tend to structure their solutions – which confirms the basic assumption that these regions might represent their concepts in mind. In well-designed specifications they have to exist, otherwise any structuring technique will fail.

ad (b) A simple experiment can show the attainable reduction with specificatoins of raising sizes. For the specifications (including the ICT window manager) all possible regions (considering different combinations of dependencies) can be generated. Fig. 8.10 demonstrates the achieved reductions. The mean decreases from 0.528 (for a simple specifications with 24 primes) to 0.218 for the ICT Window Manager specification (with 213 primes). In addition to that it also shows that the reduction works better with the growing sizes. One might expect that, when the size of a specification is doubling, the estimated sizes of the resulting regions might be doubling. It is even better, and Fig. 8.10 demonstrates this by comparing the factor of the estimated sizes to the real ones. For the ICT window manager the reduction is $2.68\ (= 1/0.373)$ times better than expected.

The reduction factor depends on several aspects. The compactness of a specification and the interrelationship between primes not being the least among them. Nevertheless, the analysis suggests that the effect obtainable rises with the size of specifications under consideration. The regions stay at manageable size.

What remains is the questions concerning the time complexity of the underlying algorithms. In fact, the methods in the background are efficient enough to deal with typically available formal specifications. The fuzzy clustering approach has a run-time complexity of $O(n)$, and, as we are pre-calculating all $C = (n-1)$ clusters, we get a run-time complexity of $O(n^2)$. The pattern vector calculation is the crucial and time-consuming part. However, the slicing/chunking framework [4] is based on reachability arguments and thus already computes the necessary values in $O(n^2)$. For the *Elevator* specification it took about 100 seconds on a Pentium 4, 3.2 GHz, 1GB RAM, to perform the ASRN transformation, cluster generation, and indices calculation – still faster than reading the whole specification.

8.6 Conclusion

This paper introduces the notion of a formal specification concept and presents a process model for concept location. The basic idea is to transform the specification into a directed graph (where prime objects are represented by vertices), and thus facilitate the generation of partial specifications (clusters, slices/chunks). Concepts are identified by looking for

slices, chunks, and clusters which are defined in such a way that those primes are regarded that are related to other primes in respect to specific types or numbers of dependencies.

The approach of specification concept identification is yet at the beginning and the next steps will be as follows: firstly, much more specifications will have to be examined to proof the usability of the model. Secondly, the pattern vector is crucial and at the moment all dependencies have the same weight. Adjusting the clusters by using weighting might make the approach more flexible. Finally, an empirical study will have to be conducted in order to proof its applicability by users other than the author of the workbench.

Specifications are complex buildings, but they are no lost cause. It is the author's belief that, by using the most suitable approaches available, several myths mentioned in the introduction will have to be re-written.

Acknowledgement

Many thanks to Abdel-Hamid Bouchachia and Marcus Hassler for intensive discussions on clustering and the topic of cluster validation techniques. I am also very grateful for the remarks of Roland Mittermeir and the anonymous reviewers of the ICSM'06 program committee, who's constructive remarks motivated me to further elaborate that topic.

REFERENCES

[1] J.C Bezdek, R. Ehrlich, and W. Full. The fuzzy c-means clustering algorithm. *Computers and Geoscience*, 10(2):191–203, 1984.

[2] Andreas Bollin. *Specification Comprehension – Reducing the Complexity of Specifications*. PhD thesis, Universität Klagenfurt, April 2004.

[3] Andreas Bollin. The Efficiency of Specification Fragments. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering*, 2004.

[4] Andreas Bollin. Maintaining formal specifications. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary*, pages 442–453, 2005.

[5] Jonathan Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press (ITCP), 1996.

[6] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, July 1995.

[7] Ilene Burnstein, Katherine Roberson, Floyd Saner, Abdul Mirza, and Abdallah Tubaishat. A role for chunking and fuzzy reasoning in a program comprehension and debugging tool. In *TAI-97, 9^{th} International Conference on Tools with Artificial Intelligence*. IEEE press, November 1997.

[8] Ilene Burnstein, Floyd Saner, and Yachai Limpiyakorn. Using an artificial intelligence approach to build an automated program understanding/fault localization tool. In *11th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'99)*, pages 69–76. IEEE Computer Socienty, 1999.

[9] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[10] Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 241–247. IEEE Computer Society, 2000.

[11] Antoni Diller. *Z - An Introduction to Formal Methods*. John Wiley and Sons: Buffins Lane, Chichester, England, 1999.

[12] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. On clustering validation techniques. *Journal of Intelligent Information Systems*, 17(2-3):107–145, 2001.

[13] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[14] Mehdi T. Harandi and Jim Q. Ning. Knowledge-based program anaylsis. *IEEE Software*, 7(1):74–81, January 1990.

[15] Wojtek Kozaczynski, Jim Ning, and Andre Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, December 1992.

[16] Roland T. Mittermeir and Andreas Bollin. *Demand-Driven Specification Partitioning*, volume 2789 of *Lecture Notes in Computer Science*, pages 241–253. Springer Berlin / Heidelberg, 2003.

[17] Tomohiro Oda and Keijiri Araki. Specification slicing in a formal methods software development. In 17^{th} *Annual International Computer Software and Applications Conference*, IEEE Computer Socienty Press, pages 313–319, November 1993.

[18] Alex Quilici, Qiang Yang, and Steven Woods. Applying plan recognition algorithms to program understanding. *Automated Software Engineering: An International Journal*, 5(3):347–372, July 1998.

[19] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *International Workshop on Program Comprehension*, pages 271–278. IEEE Computer Society, 2002.

[20] Philip E. Ross. The exterminators. *IEEE Spectrum*, pages 36–41, September 2005.

[21] J.M. Spivey. *The Z Notation*. C.A.R. Hoare Series. Prentice Hall, 1989.

[22] Axel van Lamsweerde. Formal specification: a roadmap. In *ICSE 2000 - Future of Software Engineering Track*, pages 147–159, 2000.

[23] Robert Anthony Weida. *Closed Terminologies and Temporal Reasoning in Description Logic for Concept and Plan Recognition*. PhD thesis, Graduate School of Arts and Sciences, Columbia University, 1996.

[24] Christopher A. Welty. Augmenting abstract syntax trees for program understanding. In *Proceedings of The 1997 International Conference on Automated Software Engineering*, pages 126–133. IEEE Computer Society Press, 1997.

[25] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7:49–62, 1995.

[26] Steven Woods and Alex Quilici. Some experiments toward understanding how program plan recognition algorithms scale. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'96)*, pages 21–30. IEEE, 1996.

[27] Fangjun Wu and Tong Yi. Slicing z specifications. *ACM SIGPLAN Notices*, 39(8):39–48, 2004.

Appendix A: Birthday Book

The birthday book (BB for short) out of [21] describes a simple system for administrating names and birthday dates.

First names and dates are introduced as global sets. In order to indicate the success or failure of an operation, a global type $REPORT$ is introduced.

$[NAME, DATE]$
$REPORT ::= OK \mid NOK$

The state space consists of the set of all known names, and the "database" entries for the birthday dates. The predicate ensures that only known names are in the database.

BB

$known : \mathbb{P}\, NAME$
$birthday : NAME \pfun DATE$

$known = \mathrm{dom}\, birthday$

At the beginning the database is empty.

InitBB

BB

$known = \varnothing$

There are several operations for working with the database. It is possible to Add a pair $(name, date)$ to the database, and it is possible to $Delete$ an entry from the database.

Add

ΔBB
$name? : NAME$
$date? : DATE$

$name? \notin known$
$birthday' = birthday \cup$
$\quad \{name? \mapsto date?\}$

Delete

ΔBB
$name? : NAME$

$name? \in known$
$birthday' = birthday \setminus$
$\quad \{name? \mapsto birthday(name?)\}$

To indicate the success of an operation the result OK is returned.

Success

$result! : REPORT$

$result! = OK$

With the above operation schemata the functioning system consists of successfully performed add or delete operations.

$$FunctioningDB == ((Add \land Success) \lor (Delete \land Success))^*$$

Appendix B: Elevator Specification

The Elevator specification [9] describes a simple system consisting of one elevator. The elevator reacts to button events corresponding to calls for the elevator. The call for the elevator is made on a floor. Additionally a request to stop at a specific floor can be made inside the elevator.

There are 10 floors in the system. The Elevator can move up or down and its door can be open or closed.

$$MaxFloor == 10$$
$$Direction ::= up \mid down$$
$$DoorState ::= open \mid closed$$

The state space consists of the current floor, a set of pending requests, pending up- and down-calls, the current state of the movement and the door.

Elevator

$$CurrentFloor : \mathbb{N}_1$$
$$Requests : \mathbb{P}\,\mathbb{N}_1$$
$$UpCalls : \mathbb{P}\,\mathbb{N}_1$$
$$DownCalls : \mathbb{P}\,\mathbb{N}_1$$
$$Dir : Direction$$
$$Door : DoorState$$

$$CurrentFloor \leq MaxFloor$$
$$max\ Requests \leq MaxFloor$$
$$max\ UpCalls \leq MaxFloor$$
$$max\ DownCalls \leq MaxFloor$$

Initially, the elevator is at the first floor, and the door is open. There are no pending requests and calls.

InitElevator

$$Elevator$$

$$CurrentFloor = 1$$
$$Requests = \varnothing$$
$$UpCalls = \varnothing$$
$$DownCalls = \varnothing$$
$$Dir = up$$
$$Door = open$$

There are two passenger events that are to be handled. Firstly, a passenger might request to stop the elevator at a specific floor from inside the elevator ($ElevatorButton\ Event$). Secondly, a passenger calls for the elevator ($FloorButtonEvent$).

$$
\begin{array}{|l}
\hline
ElevatorButtonEvent \\
\Delta Elevator \\
Floor? : \mathbb{N}_1 \\
\hline
Floor? \leq MaxFloor \\
CurrentFloor' = CurrentFloor \\
Requests' = Requests \cup \{Floor?\} \\
UpCalls' = UpCalls \\
DownCalls' = DownCalls \\
(Dir' = Dir) \land (Door' = Door) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
FloorButtonEvent \\
\Delta Elevator \\
Floor? : \mathbb{N}_1 \\
CallDir? : Direction \\
\hline
Floor? \leq MaxFloor \\
CurrentFloor' = CurrentFloor \\
(CallDir? = down) \Rightarrow \\
\quad (UpCalls' = UpCalls) \land \\
\quad (DownCalls' = DownCalls \cup \{Floor?\}) \\
(CallDir? = up) \Rightarrow \\
\quad (UpCalls' = UpCalls) \land \\
\quad (DownCalls' = DownCalls \cup \{Floor?\}) \\
Requests' = Requests \\
(Dir' = Dir) \land (Door' = Door) \\
\hline
\end{array}
$$

$$
\begin{aligned}
PassengerEvent == {} & ElevatorButtonEvent \\
& \lor\ FloorButtonEvent
\end{aligned}
$$

In the above operation schema $FloorButtonEvent$ the bug is hidden. Instead of

$$
\begin{aligned}
& (CallDir? = up) \Rightarrow \\
& \quad (DownCalls' = DownCalls) \land \\
& \quad (UpCalls' = UpCalls \cup \{Floor?\})
\end{aligned}
$$

the specification contains the following (buggy) lines:

$$
\begin{aligned}
& (CallDir? = up) \Rightarrow \\
& \quad (UpCalls' = UpCalls) \land \\
& \quad (DownCalls' = DownCalls \cup \{Floor?\})
\end{aligned}
$$

The movement of the elevator is complex. An elevator might move up or down ($BasicMoveUp$ or $BasicMoveDown$) when it is servicing requests or calls.

$$
\begin{array}{|l}
\hline
BasicMoveUp \\
\Delta Elevator \\
\hline
Dir = up \\
Requests \cup UpCalls \neq \varnothing \\
CurrentFloor \leq max\ (Requests \\
\qquad \cup UpCalls) \\
CurrentFloor' = min\ \{x : \mathbb{N}_1 \mid x \in \\
\qquad (Requests \cup UpCalls) \\
\qquad \land\ x > CurrentFloor\} \\
Requests' = Requests \setminus \\
\qquad \{CurrentFloor'\} \\
UpCalls' = UpCalls \\
DownCalls' = DownCalls \setminus \\
\qquad \{CurrentFloor'\} \\
(Dir' = up) \land (Door' = Door) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
BasicMoveDown \\
\Delta Elevator \\
\hline
Dir = down \\
Requests \cup DownCalls \neq \varnothing \\
CurrentFloor \leq \\
\qquad min\ (Requests \cup DownCalls) \\
CurrentFloor' = max\ \{x : \mathbb{N}_1 \mid x \in \\
\qquad (Requests \cup DownCalls) \\
\qquad \land\ x < CurrentFloor\} \\
Requests' = Requests \setminus \{CurrentFloor'\} \\
UpCalls' = UpCalls \setminus \{CurrentFloor'\} \\
DownCalls' = DownCalls \\
(Dir' = down) \land (Door' = Door) \\
\hline
\end{array}
$$

However, when all up-calls and requests above the current floor have been serviced and there are still pending calls and requests, the elevator has to start moving downwards ($ChangeUpToDown$). When all down-calls and requests below the current floor have been serviced and there are still pending calls and requests, the elevator has to start moving up ($ChangeDownToUp$).

$$
\begin{array}{|l}
\hline
ChangeUpToDown \\
\Delta Elevator \\
\hline
Dir = up \\
(Requests \cup UpCalls \neq \varnothing \land \\
CurrentFloor > \\
\quad max\ (Requests \cup UpCalls)) \lor \\
Requests \cup UpCalls = \varnothing \\
Requests \cup DownCalls \neq \varnothing \\
CurrentFloor' = \\
\quad max\ (Requests \cup DownCalls) \\
Requests' = Requests \setminus \\
\qquad \{CurrentFloor'\} \\
UpCalls' = UpCalls \\
DownCalls' = DownCalls \setminus \\
\qquad \{CurrentFloor'\} \\
(Dir' = down) \land (Door' = Door) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
ChangeDownToUp \\
\Delta Elevator \\
\hline
Dir = down \\
(Requests \cup DownCalls \neq \varnothing \land \\
CurrentFloor < \\
\quad min\ (Requests \cup DownCalls)) \lor \\
Requests \cup DownCalls = \varnothing \\
Requests \cup UpCalls \neq \varnothing \\
CurrentFloor' = \\
\quad min\ (Requests \cup UpCalls) \\
Requests' = Requests \setminus \{CurrentFloor'\} \\
UpCalls' = UpCalls \setminus \{CurrentFloor'\} \\
DownCalls' = DownCalls \\
(Dir' = up) \land (Door' = Door) \\
\hline
\end{array}
$$

When all requests and calls have been serviced the elevator checks if there are new calls. If there are up-calls then it restarts moving upward ($RestartMovingUp$). If there are down-calls then it restarts moving down ($RestartMovingDown$).

RestartMovingUp

$$\Delta Elevator$$

$$Dir = up$$
$$UpCalls \neq \varnothing$$
$$CurrentFloor > max\ UpCalls$$
$$DownCalls \cup Requests = \varnothing$$
$$CurrentFloor' = min\ UpCalls$$
$$Requests' = Requests$$
$$UpCalls' = UpCalls \setminus \{CurrentFloor'\}$$
$$DownCalls' = DownCalls$$
$$(Dir' = up) \land (Door' = Door)$$

RestartMovingDown

$$\Delta Elevator$$

$$Dir = down$$
$$DownCalls \neq \varnothing$$
$$CurrentFloor < min\ DownCalls$$
$$UpCalls \cup Requests = \varnothing$$
$$CurrentFloor' = max\ DownCalls$$
$$Requests' = Requests$$
$$UpCalls' = UpCalls$$
$$DownCalls' = DownCalls \setminus \{CurrentFloor'\}$$
$$(Dir' = down) \land (Door' = Door)$$

$$Move == BasicMoveUp \lor BasicMoveDown \lor ChangeUpToDown \lor ChangeDownToUp \lor RestartMovingUp \lor RestartMovingDown$$

The state of the door is important, too. It can be opened, and in case of an request or call, it is closed.

OpenDoor

$$\Delta Elevator$$

$$CurrentFloor' = CurrentFloor$$
$$Requests' = Requests$$
$$UpCalls' = UpCalls$$
$$DownCalls' = DownCalls$$
$$Dir' = Dir$$
$$Door' = open$$

CloseDoor

$$\Delta Elevator$$

$$Requests \cup DownCalls \cup UpCalls \neq \varnothing$$
$$CurrentFloor' = CurrentFloor$$
$$Requests' = Requests$$
$$UpCalls' = UpCalls$$
$$DownCalls' = DownCalls$$
$$Dir' = Dir$$
$$Door' = closed$$

When there are no requests the elevator stays at the current floor.

NoRequestsOrCalls

$$\Xi Elevator$$

$$Requests \cup UpCalls \cup DownCalls = \varnothing$$

The elevator repeatedly closes the door, moves, and opens the door. Between two events it may receive passenger events.

$$MoveCycle ==$$
$$(CloseDoor \fatsemi PassengerEvent^{\circ}_{9}$$
$$Move \fatsemi (PassengerEvent)^{\circ}_{9}$$
$$OpenDoor \fatsemi PassengerEvent)^{*}$$
$$ElevatorCycle ==$$
$$(MoveCycle \lor NoRequestsOrCalls)^{*}$$

$$FunctioningElevator ==$$
$$(PassengerEvent \fatsemi ElevatorCycle)^{*}$$

Appendix C: Partial SRN of the Elevation Specification

Fig. 8.11 shows the ASRN of the elevator specification of App. B. The graph represents the 32 higher-level primes an their syntactical interrelationship.

The output has been generated by *dotty* (see www.graphviz.org for a description of the tool) and by doing an auto-layout. Thus no re-arrangement of the vertices has been made.

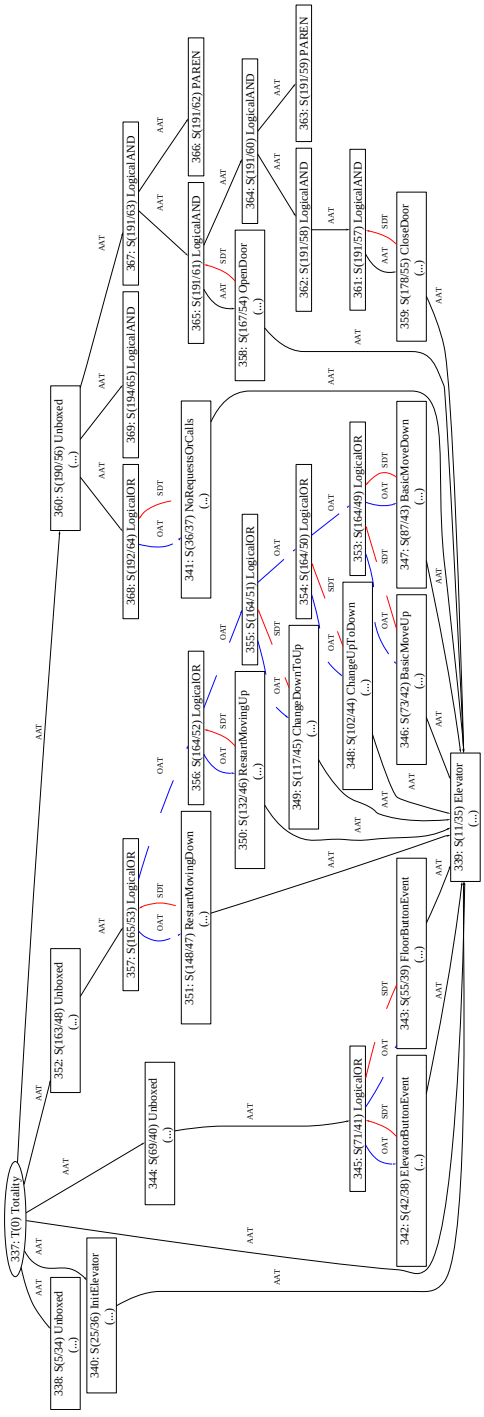


Figure 8.11 The *Elevator* specification out of [9]. Please note that for reasons of space several dependencies are omitted (logical combinations are resolved by AAT, AND-arc types and OAT, OR-arc types).

CHAPTER 9

CONCEPT MANAGEMENT: IDENTIFICATION AND STORAGE OF CONCEPTS IN THE FOCUS OF FORMAL Z SPECIFICATIONS

D. Pohl, A. Bollin

Communications in Computer and Information Science. Springer, 69(II):248-261, 2010.

Abstract

Concept location is a necessary but all too often laborious task during maintenance phases. Part of the reasons is that repeatedly the same or similar concepts have to be reconstructed, which is a resource and time-consuming process. This contribution investigates the situation and suggests a framework that persistently stores conceptual elements and their dependencies in an SQL database. On the example of formal Z specifications it demonstrates that concept location is alleviated by simple queries that automatically identify concepts based on the database entries.

9.1 Introduction

"*People like to write code, but they do not like to read somebody else's code.*" This statement becomes increasingly apparent as the number of software systems in use is growing – and have to be maintained. Why might this be the case?

In [21, p.242] it is postulated that it is easier to express ones owns concepts and ideas into the tight formality of a (programming) language than to reconstruct the concepts the original developer had in mind from the formal expressions formulated in low level code.

This observation is above all true when the text or code expresses a concept previously unknown to the reader – which is often the case in maintenance situations. In the lucky case there are at least high-level specification documents around, but, without supporting tools, the identification of the relevant information stays a hard business.

Maintenance activities are often formulated in terms of adding/changing/deleting features or concepts [33], and concept location techniques play an important role, in software as well as in specification maintenance. Formal specification frameworks provide excellent support for editing and verification, but they do not provide concept location facilities. A (semi-) automatic identification of concepts is missing and, for formal specifications, also the possibility to store the, often cumbersome, reconstructed concepts to be found in the documents.

The objective of this contribution is to demonstrate that not so much has to be done in order to identify *and* store concepts. We introduce a generic model that is able to deal with documents and concepts of different types. As a proof of concept a prototype for formal Z specifications [30] has been implemented. But the basic ideas also apply to other artifacts ... from natural language descriptions to program code.

The paper is structured as follows. Sec. 9.2 discusses the related work. Sec. 9.3 derives the requirements for a framework that is able to persistently store identified concepts. With the necessary basics in concept location of Sec. 9.4 in mind, Sec. 9.5 introduces the key ideas behind our suggested framework. Sec. 9.6 evaluates its functionality in respect to correctness and time complexity. The contribution closes with a short summary and an outlook of work to be done.

9.2 Related Work

Due to the size of today's systems, maintenance and reverse engineering activities are usually supported by tools and frameworks.

At first, there are SW-Engineering frameworks that can also be used for reverse engineering activities [29, 12, 23, 17, 20]. Usually, they enable the reconstruction or extractions of diagrams from code, and thus establish links between different representations supporting round-trip engineering. Their disadvantage is that they are limited to a very specific notation (e.g. UML) or assume that the code has already been written within the framework.

Another group is that of explicit reverse engineering tools. There, the input is the code or an abstract representation of it, and they sustain the process of creating extractions or views onto the source. Popular representatives are RIGI [22], going back to the work of Müller, Tilley, and Wong in the early 90s, or Bauhaus [19]. In the meantime there are a lot of extensions and, especially for C++ and Java programs, similar frameworks [14, 18, 10, 16, 11, 15].

Finally, there are frameworks that focus explicitly on concept location [9, 34, 27]. They make use of techniques similar to those of reverse engineering environments, but provide additional support for storing and retrieving previously identified concepts.

Environments for formal specifications have their focus on writing down a syntactically correct specification and providing verification support. They are not meant to be used for reverse engineering. However, one tool-set that permits looking at a specification from different angles is $VDMTools$ with its *RoseLink* feature [1]. It can be used to generate UML

diagrams from VDM specifications. Tools for B also focus on the creation of the specification. Some representatives, e.g. Atelier-B [13], at least provide views onto dependencies between the components. In the case of Z the situation is almost the same. One exception is the ViZ toolkit [5], where the emphasis was laid on concept location. But ViZ also has its limitations, first and foremost the inability to persistently store identified concepts.

9.3 Maintenance Support

The motivation for this work goes back to a project where we tried to improve maintenance and re-engineering activities of formal Z specifications. A big advantage of formal specifications is their semantic density. One can express his or her thoughts precisely and on a high level of abstraction. But with that, the complexity (and density) of even small specifications is quite high. As shown in [4], specifications might have thousands of dependencies between their elements, and comprehension aids are definitely necessary.

9.3.1 RE of Formal Z Specifications

At first sight approaches from the traditional field of software comprehension are not suitable. Z (among others) is a state-based, declarative specification language, with no explicit control and data flow – dependencies that are typically utilized when looking for concepts. But there is a solution to this problem.

In [3, p.60–63] a syntactical approximation to the identification of dependencies was described, which then enabled the identification of concepts like slices, chunks, and clusters within formal specifications. ViZ (for Vizualization of Z Specifications) implements these algorithms and supports typical comprehension activities. But with its employment the following issues have been observed:

- The same comprehension steps are often carried out more than once - even if there is additional documentation. So dependencies and concepts identified once have to be reconstructed again.

- The calculation and identification of concepts is still a time consuming task. Moreover, these calculations and findings are lost when the framework is closed and the state is not saved for later use.

- It is not sufficient to look at a concept in isolation. Depending on the problem at hand more than just a single view onto the artifact has to be generated.

To summarize, a framework sustaining comprehension tasks should not ignore the above observations. It should support the identification of new concepts at different levels, enable the linkage between them, and be able to store the findings persistently in a database. Please note that the observations above are not only limited to the field of Z specifications. They apply to other artifacts, too. Due to the resource-consuming calculations necessary for dense and complex formal specifications, the storage of concepts is of major importance in our case.

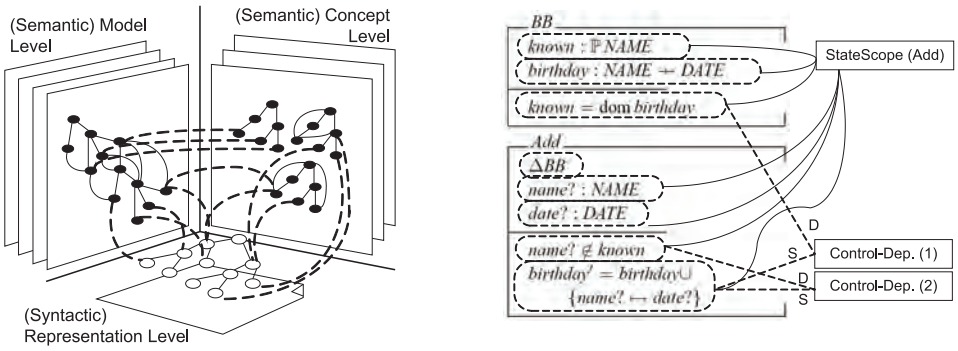


Figure 9.1 (Left) An artifact contains concepts in three different dimensions and at different levels. (Right) Specifications are dismantled into syntactical elements (primes) and then extended by dependency and scope annotations.

9.3.2 Multi-dimensional Problem

The reconstruction of concepts within an artifact is not trivial. In order to reconstruct (or better approximate) the concepts a former developer had in mind, one has to take into account that different facets led to the writing of the artifact:

- *The environment and context of the problem.* There are maybe several assumptions the developer had to consider and that are not fully documented. So, an artifact only makes sense when put into the right context.

- *The concepts inherent in the language.* Different (programming) languages are differently suitable for describing problems. In fact, the semantics behind a language is often used to reduce writing effort. E.g. dependencies do not have to be made explicit, names are declared once, and it is clear when they are usable and when not. Those concepts, let us call them *"behind the scenes"*, are important for grasping the whole meaning.

- *Concepts made explicit in the source.* The concepts mentioned above are problem- and language-inherent. What is left are those concepts that are visible in the artifact. E.g. a case-statement represents an n-ary decision, and its meaning is clearly defined. Such concepts are called *"before the scenes"*.

When one is at least familiar with the problem field and the environment, the identification of the concepts behind and before the scenes might be seen as a multidimensional problem. Fig. 9.1 (left side) demonstrates this viewpoint.

The **(Syntactic) Representation Level**. At first, there is the artifact itself. It has been written in some pre-defined language, with clearly defined rules for its syntax. It is assumed that it can be divided into a structured set of basic elements (primes, statements, paragraphs ...). This sequence of elements, its nouns and verbs, and the structure make up a lot of the underlying concept(s). Thus, the representation level deals with the source and the structure of the artifact.

The **(Semantic) Model Level**. In general, a language comes along with a clearly defined semantics (e.g. statements have to be put into some order). This implies a specific meaning and leads to several dependencies and relations between the elements (see Fig. 9.1). These rules are not written down in the artifact, but belong to it and make up another part of the underlying concept(s). They can be seen as named concepts, going back to the semantic possibilities of the language at hand.

The **(Semantic) Concept Level**. What is left are the concepts the developer expresses unconsciously. They describe specific aspects of the problem and are recognizable when looking at the artifact from some distance. These mental macros, as Baxter et. al. [2] call them, express higher-level concepts, and program comprehension techniques are typically used to carve them out. To this dimension belong concepts like slices [31], chunks [7], clichés [6], and different types of clusters [32].

To exemplify the situation, the calculation of a specification or program slice (stored in the concept level) depends on the concept of dependencies (model level), the concept of scope (model level), and the basic elements in the source (representation level). When these concepts (at different levels) are calculated they can be stored in a database for later use.

9.4 Formal Specification Concepts

The model presented above has been mapped to a database schema and forms the basis for the concept location process. Though the strength of the framework is to deal with different types of documents, our experiences arise from the scope of maintaining formal Z specifications – which also was the starting point of the requirements' considerations. The specification concepts we are interested in are those as described in more detail in [5]: slices, chunks, and clusters.

9.4.1 Conceptual Elements

A *formal specification concept* is a coherent, abstract (or generic) pattern of specification text that is generalized from particular instances of the specification. It can be understood and recognized as a whole even when standing alone.

As explained in more detail in [5, p.81], the basic elements such concepts are built upon are called specification primes. Such *formal specification primes* (also called prime-objects) also represent the basic entities of a specification. They are built from literals of the specification and form logic, syntactic, or semantic units. A prime is a syntactically coherent sequence of literals within a specification, forming semantic entities that can be paraphrased by a short sentence in natural language.

With programming languages, primes would be programming statements. In the case of formal Z specifications these primes are declarations, definitions, and predicates. Fig. 9.1 (to the right) marks the primes by dashed ellipses, e.g. the prime "$name? \notin known$" (the second prime from below).

When primes are combined they do form so-called *higher-level primes*. The Add schema operation in Fig. 9.1 is an example of such a higher-level prime, telling the user about the things happening when the operation applies.

9.4.2 Specification Concepts

Concepts within formal specifications are identified in an iterative manner [5, p.83]. Starting with a domain-level request, one forms a mental model of the problem in mind and concept location is about to begin. Concept candidates are identified and matched against the model of the problem. When the candidates match, the concept is (very likely) identified and the elements of the related candidates are tagged. The concept location process makes use of the following steps: pattern matching, slicing and chunking, and cluster identification.

As explained in [28], experienced users first browse the text and try to identify relevant parts by *grep*-ing for keywords. When this is not successful, more complicated methods are used. Structures are especially of interest, and clustering is a feasible way in identifying related regions. The selection might be based on the use of identifiers, or on the number of dependencies that glue the primes together. Similarly, slices and chunks can be generated for a point of interest by just looking at specific primes and by following different types of dependencies.

Specification concepts are identified by looking at dependencies among primes. For the calculation of slices, chunks, and clusters, control- and data-dependencies are needed, and though these dependencies are not explicitly available, they can be reconstructed by looking at pre- and post- conditions. The approach goes back to the work of Oda and Araki [24] and has been refined in [8, 3]. The basic idea is that, within a specific scope, primes that are part of post-conditions are dependent on primes that contribute to pre-conditions. In order to ease their identification, all primes get tagged with annotations. For every identifier used in a prime the following meta-information is assigned to the prime: CI (channel input) when it is an input identifier which is decorated by a $\boxed{?}$, CO (channel output) for an result identifier decorated by an $\boxed{!}$, D (declaration) for an identifier that denotes the identifier's after-state and which is decorated by a $\boxed{'}$, T (type declaration) for identifiers that are declared, and U (used) otherwise. So, the two Z primes (of the Add schema in Fig. 9.1)

$$\begin{array}{ll} P1: & name? \notin known \\ P2: & birthday' = birthday \cup \{name? \mapsto date?\} \end{array}$$

would be tagged as follows. Prime $P1$ is annotated by $\{CI \mapsto \{name\}, U \mapsto \{known\}\}$, and prime $P2$ is annotated by $\{D \mapsto \{birthday\}, U \mapsto \{birthday\}, CI \mapsto \{name, date\}\}$. Post-condition primes are those primes that have a tag containing a D or CO set. In our case $P2$ would be a post-condition prime, prime $P1$ is a so-called pre-condition prime.

The identification of dependencies is explained in more details in [3, pp.126–132]. However, when the meta-information is stored in the database (and assigned to the prime objects), the queries are quite simple. Our agents, as introduced in Sec. 9.5.2, make use of this meta-information in form of SQL queries.

9.5 Concept Location Framework

The framework for identifying the different concepts in Z specifications is designed to cope with different types of artifacts. It implements a traditional client-server architecture pattern based on the EJB Technology. The client is responsible for visualizing the results

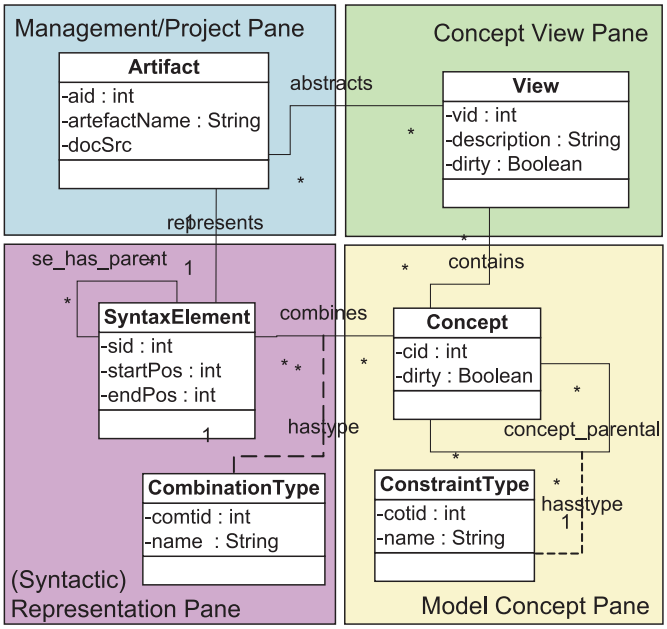


Figure 9.2 The four different panes of the database model. (Please note that for reasons of space only the major entities are shown. See [25] for more details.)

and for triggering the concept extraction. On the server side it is designed to handle different types of artifacts. Whenever a document is stored, different analysis tasks are started by a scheduler extracting concepts, and the findings are stored in the database again (see Sec. 9.5.2).

9.5.1 Database

The database forms the basis for the management of conceptual elements and their dependencies, sustaining the concept location process of formal Z specification documents.

There are four areas covered by the database. Three of these areas are related to the multi-dimensional view as described in Sec. 9.3.2. The forth area is used for the management of artifacts within the software engineering life-cycle.

9.5.1.1 Management/Project Pane. Based on the software engineering process, the $Management/Project$ section deals with the management of artifacts within different phases of the project. There, a $Project$ consists of different $Phases$. Within each phase $Artifacts$ are created, most of them depending on each other. Different artifact versions might exist. Hence, the database schema takes this into account by assigning the $ArtifactMetaData$ information to an artifact.

9.5.1.2 (Syntactic) Representation Pane. Every artifact, independently of its nature, consists of a certain structure. This structure is built upon elements that are called $Syntax\ Element$s. $SyntaxElements$ are characterized by their $ElementType$s:

- *Content:* It represents a pure structural element (so-called basic elements like sentences, expressions, or statements).

- *Presentation:* Text is often decorated (e.g. by boxes). As sometimes this decoration carries information, it is also stored.
- *Aggregation:* It provides the possibility to explicitly mark higher-level concepts that have been created by the aggregation of basic elements.

Syntax elements carry a lot of information. E.g. they refer to identifiers, define labels, or describe some input operations. A set of meta-data is introduced to store them. Every data entry of $ElementMetaData$ belongs to a specific $AnnotationType$, and so different (but consistent) categorizations get possible.

9.5.1.3 Model Concept Pane. A $Concept$ corresponds to one or several syntactical elements (*SyntaxElements*). For different types of concepts also different relationships are possible. This is done by the *CombinationType* entity. Besides, it is possible to express some kind of direction or ordering between the related elements. The characterization of a concept is made up by the $ConceptType$ entity. Concepts also form hierarchies, and to express these relations, an n-to-m recursive relationship is introduced.

9.5.1.4 Concept View Pane. The database schema allows for different views onto the concepts, be them explicitly or implicitly available. The main purpose of the $View$ entity is to cluster related concepts (concepts of the same type) or to form different views onto the current artifact. This information is, besides the creation date, stored in the $ViewMetaData$ entity. Every view belongs to a certain category. This classification is stored in the $ViewType$.

It is also possible to annotate a view with $ViewData$ entries that are of specific $View$ $DataType$s. Those entries are, e.g., used to store metrics of clusters or other characteristics relevant for concepts within the view. These steps are carried out by agents like those introduced in the following section.

9.5.2 Agents

Our prototype provides different agents: scope agents that regard scope rules of Z, dependency agents for reconstructing dependencies, and, based on them, slice/chunk/cluster agents for carving out higher-level concepts.

For the creation of slices or chunks typically two different types of dependencies (data-, and control-dependencies) and the syntactical environment are necessary. So, at first, these dependencies have to be extracted, but the extraction is complicated due to language-specific scope rules. The first task for our framework is therefore to reconstruct the concepts representing the scope.

In the context of formal Z specifications three types of scopes can be identified (and are calculated by three agents in our framework). The *Declaration Scope* represents all visible declarations for a prime in the specification. The scope is also needed to derive the syntactical dependencies and thus for building syntactically correct partial specification. The *State Scope* deals with schema inclusions within a specification document and aggregates the primes of the inclusion and the primes of the including schemata (see Fig. 9.1, $Add_{(State)}$ for an example). Finally, the *Connectivity Scope* merges all primes of two or more schemata combined via schema operations.

In our framework, at first, the agents launch queries to identify the correct scopes, then they start reconstructing control- and data dependencies. Sec. 9.5.3 demonstrates the simplicity on the example of control dependency identification.

After scope and dependency calculation the *Slice* and *Chunk* agents can be activated by the user. Beginning with a "point of interest" (a set of primes), the agents calculate slices and chunks by following the stored control- and data dependencies. The results are again stored in the database for later use.

The last agent presented here is called *Clustering Agent*. It is responsible for the generation of clusters of a specification document. In order to ease deciding about the most useful number of clusters to be generated, the agent pre-calculates and stores all variants of them. Every cluster view is then extended by meta-information. This meta-data describes different types of cluster-metrics, like the partition entropy or the partition coefficient measure. This information can later help the user to decide about the usefulness of the clusters.

Some of the agents are executed in parallel; other agents have to wait. Therefore all agents are registered in an agent scheduler, which is responsible for the right execution order.

9.5.3 Queries for Concept Location

Our framework makes use of a simple idea: the calculation logic is moved from traditional program code to SQL queries disposed by the agents. The extraction is done by expressions which are based on the annotations of the primes in the database. For Z documents the necessary queries are already implemented.

To demonstrate the elegance of the queries we look at the steps necessary to carve out control dependencies from Z specifications. The relevant primes in the database are the syntax elements tagged by the $Content$ element type. The extraction-process then uses the *State* and *Connectivity Scope* for the calculation.

$$\begin{aligned}
&\Pi_{sid}\ \sigma_{AnnotationType.name=\text{"}D\text{"}} \\
&((\sigma_{Concept.id=act}\, Concept \bowtie \\
&(\sigma_{ConceptType.name=\text{"}State\text{"}}\, ConceptType)) \\
&\bowtie SyntaxElement \bowtie \\
&ElementMetaData \bowtie AnnotationType)
\end{aligned} \tag{9.1}$$

$$\begin{aligned}
&\Pi_{sid}(\sigma_{Concept.id=act}\, Concept \bowtie \\
&(\sigma_{ConceptType.name=\text{"}State\text{"}}\, ConceptType)) \\
&[sid \neq sid] \\
&\Pi_{sid} \\
&(\sigma_{\substack{AnnotationType.name\neq\text{"}T\text{"}\ or \\ AnnotationType.name\neq\text{"}C\text{"}\ or AnnotationType.name\neq\text{"}D\text{"}}} \\
&SyntaxElement \bowtie ElementMetaData \\
&\bowtie AnnotationType)
\end{aligned} \tag{9.2}$$

Spec.	Lines	Pages	Primes	CD	DD	ViZ	DB	Concepts
BB	40	2	34	10	5	4.6	7.0	36
Cinema	95	4	74	121	43	75.3	43.2	114
Petrol	88	3	65	192	177	152.9	51.9	219
Elevator	193	6	185	1,628	992	1,223.4	709.3	984

Table 9.1 Complexity attributes (lines, A4 pages, number of primes, control- and data dependencies), calculation times (needed by the *ViZ* component, and the EJB database interface) in seconds, and number of identified concepts.

The queries (3.1) and (3.2) extract control-dependencies of the Add schema operation of the *'Birthday Book'*-Specification (where act represents the identifier of the current $State\ Scope$). The queries lead to the source (S) and destination (D) primes for the dependency arcs. In fact, the results of the queries are elements of the $SyntaxElement$ entity. The agent takes all elements resulting from the first query and connects them to the resulting elements of the second query. Every pair forms a $Concept$ within the database. This information is stored in the database and results in the concept entries *Control-Dep. (1)* and *Control-Dep. (2)*) as exemplified in Fig. 9.1.

The identification of data-dependencies is similar to that of control-dependencies. Its only difference is related to the U tag, and the consideration of the name of an identifier. A detailed description of the queries for scope and dependency calculation can be found in [25, p.102-106].

9.6 Evaluation

The evaluation of the framework was carried out in two steps. First, the correctness of the identified concepts were checked, and, secondly, the usefulness in respect to performance explored. In fact, both steps also hearken back to results we gained from the existing ViZ framework.

9.6.1 Setting and Correctness

The first step was the validation of the concepts that have been identified by the agents and stored in the database. The evaluation involves specifications of raising sizes, known as Birthday Book [30], Petrol Station [3], and Elevator [8]. Additionally, a student's specification (Cinema) was added to the set. Tab. 9.1 (left part) summarizes the key attributes in order to assess the complexity of the specifications. It exemplifies the number of lines, pages (when printed), primes, control- (CD), and data dependencies (DD).

The correctness of the identified concepts was checked in two steps. At first, the new framework was used to identify dependencies, slices, and chunks. The results were then exported to a structured file. In a second step these entries have been compared to the concepts and dependencies identified by the ViZ framework. As Tab. 9.1 (right column) demonstrates, this involved 1353 concepts (consisting of slices and chunks for every prime occurring in the predicate part of every schema) and 3168 dependencies (CD and DD).

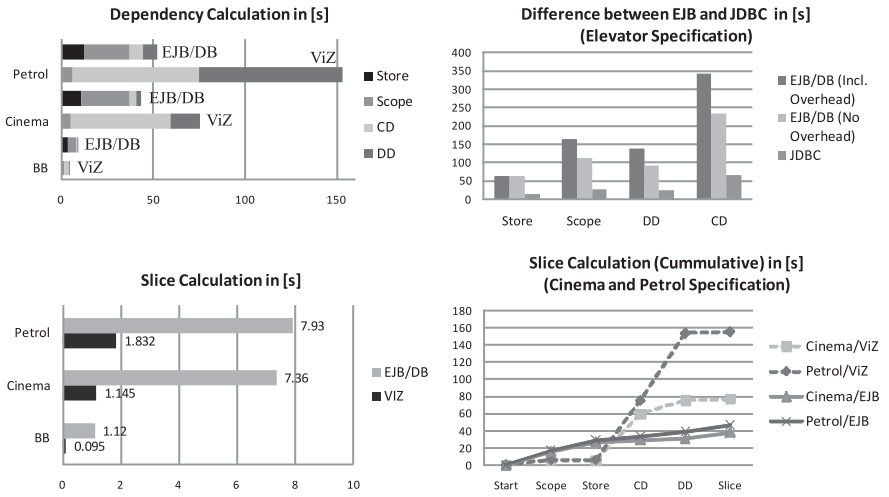


Figure 9.3 Performance considerations between the ViZ and the EJB framework.

9.6.2 Performance Considerations

As every dependency and concept has been detected correctly, we were also eager to see whether the framework scales and improves operating speed. In fact, in our case operation time it up to (a) dependency calculation, (b) storage and retrieval, and (c) concept identification.

In the case of ViZ the calculation of dependencies (and thereinafter slices or chunks) is time-consuming. ViZ uses an annotated graph to store primes and its connections, and dependency calculation is based on reachability considerations. It has a runtime performance of $O(n * (m + n * log * n))$ (with n related to the number of primes and m related to the number of dependencies in the specification). The new framework considers Def-Use equations based on scope relations (that are stored in the database), and its runtime complexity is in $O(n^2)$. Tab. 9.1 (center part) presents the time needed to calculate all dependencies for the ViZ environment and the EJB based framework (where the system consisted of a Windows XP Professional OS, Intel Core2 CPU, 2.00GHz, 2 GB RAM). This difference can also be seen in Fig. 9.3 (top left). Though ViZ does not store the elements in a database, the total time is much higher due to the extra time needed for control and data dependency calculation.

The type of the database access is also crucial for the performance. The most complex artifact in our considerations is the $Elevator$ specification, and it takes about ten minutes till all dependencies are analyzed (and about 2,600 data-sets are stored for later use). As a few thousand data-sets are not so much for a database, we were eager to know why it took so long.

It turned out that a lot of time is lost due to EJB's synchronization between the database and Java's objects. The overhead is about *one-third* of the time. Furthermore, there is very high execution time latency between EJB and its corresponding $JDBC$ queries. As explained in more details in [26, p.234], in our setting $JDBC$ scales about *six times* better than EJB. Fig. 9.3 (top right) demonstrates this time-differences on the example of the elevator specification.

The calculation of concepts like slices or chunks implies looking at a specific prime and following the relevant dependencies. Fig. 9.3 (bottom left) shows that ViZ is definitively faster than the new environment. There, all possible slices for three different specifications have been generated once and the total time measured. ViZ is much faster, which is not surprising as it's internal graph already contains the dependencies as arcs and they do not have to be read from a database. However, the new framework stores the slice as a view for later use, and calculated once, it does not have to be (re-)calculated again.

Considering the above observations, the new framework seems to be an improvement in the case of concept location environments for Z specifications. Fig. 9.3 (bottom right) demonstrates this by accumulating the time till all possible slices have been calculated once. ViZ is faster at the beginning, as it does not store the elements in a database, but the new framework invests in storing the syntactical elements in the database and assigns scope information to it. This investment pays back when dependencies are to be calculated, and it outpaces ViZ. Retrieving the concepts then is slower, but merely depends on the number of elements to be retrieved by a select operator. In addition to that, they have only to be retrieved once, as after retrieval they are stored as a view in the database. Here the strengths of a relational database pay off.

Though the new framework is faster, we conclude from the analysis above that the use of EJB is less suitable. It brings maintenance advantages, but, as also addressed in [26], one has to expect performance loss that should not be neglected. For this reason we are currently working on a new release of the framework that replaces the middleware technology by $JDBC$.

9.7 Conclusion

This paper introduces the problem of concept location within state-based specifications and motivates for a framework that persistently stores concepts for later use and fast access. Starting with a thorough analysis of concept location aspects, a database schema has been developed which, thereinafter, forms the basis for our concept location framework for formal Z specifications.

The paper then introduces the key ideas behind our prototype. Besides storing concepts, it is based upon the idea of individual agents that quickly identify different relations among syntactical elements (of our specification) stored in the $MySQL$ database. Their elegance originates from the fact that an SQL database is very efficient in looking for specific relations between elements, and thus most of the calculation logic could be put into slim SQL queries.

The evaluation is based on a comparison with ViZ, an already existing concept location framework for Z specifications. The evaluation shows that it produces the same results than ViZ, but calculation times varied. The performance of the framework was strongly influenced by EJB. The analysis of $JDBC$ and EJB shows a high factor of performance loss when using EJB. $JDBC$ scales about six times better than EJB in terms of runtime. Additionally, EJB implements an intermediate layer and, therefore, runs into performance latencies. It is going to be replaced by $JDBC$ in the next release of our framework.

REFERENCES

[1] Sten Agerholm and Peter Gorm Larsen. *Lecture Notes in Computer Science*, volume 1641, chapter Applied Formal Methods – FM-Trends 98, pages 326–239. Springer, 1999.

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo SantAnna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance, IEEE Computer Society*, pages 368–377, 1998.

[3] Andreas Bollin. *Specification Comprehension – Reducing the Complexity of Specifications*. PhD thesis, Universität Klagenfurt, April 2004.

[4] Andreas Bollin. The Efficiency of Specification Fragments. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering*, 2004.

[5] Andreas Bollin. Concept Location in Formal Specifications. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(2):77–104, March/April 2008.

[6] Andrew Broad and Nick Filer. Applying Case-Based Reasoning to Code Understanding and Generation. In *Proceedings of the Fourth United Kingdom Case-Based Reasoning Workshop (UKCBR4)*, pages 35–48, University of Salford, Salford, England, September 1999.

[7] Ilene Burnstein, Katherine Roberson, Floyd Saner, Abdul Mirza, and Abdallah Tubaishat. A Role for Chunking and Fuzzy Reasoning in a Program Comprehension and Debugging Tool. In *TAI-97, 9^{th} International Conference on Tools with Artificial Intelligence*. IEEE press, November 1997.

[8] J. Chang and D. Richardson. Static and Dynamic Specification Slicing. In *In Proceedings of the Fourth Irvine Software Symposium, Irvine, CA*, April 1994.

[9] Kunrong Chen and Vaclav Rajlich. RIPPLES: Tool for Change in Legacy Software. In *IEEE International Conference on Software Maintenance*, page 230, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[10] Computer Human Interaction and Software Engineering Lab (CHISEL). SHriMP Homepage. www.thechiselgroup.org/shrimp, Page last visited: Oct. 2008.

[11] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO – Generic Understanding of Programs – An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[12] Eclipse-GMT. Homepage. www.eclipse.org/gmt/, Page last visited: March 2009.

[13] ClearSy System Engineering. The Atelier-B Homepage. http://www.atelierb.eu/index-en.php, Page last visited: June 2009.

[14] R. Ferenc, A. Beszedes, M. Tarkiainen, and T. Gyimothy. Columbus – Reverse Engineering Tool and Schema for C++. In *IEEE International Conference on Software Maintenance*, pages 172–181, Montreal, Canada, 2002.

[15] Ric Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. GXL Graph Exchange Library Homepage. http://www.gupro.de/GXL/, Page last visited: April 2008.

[16] Rick Holt. PBS – The Portable Bookshelf Homepage. www.swag.uwaterloo.ca/pbs, Page last visited: Oct. 2008.

[17] F. Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA 2005), Workshop on Traceability*, 2005.

[18] E. Korshunova, M. Petkovic, M. G. J. van den Brand, and M. R. Mousavi. CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code (Tool Paper). In *Working Conference on Reverse Engineering (WCRE'06)*, Benevento, Italy, 2006.

[19] Rainer Koschke. Software Visualization for Reverse Engineering. *Lecture Notes in Computer Science*, 2269:524–527, 2002.

[20] MetaEdit+. Homepage. www.metacase.com, Page last visited: March 2009.

[21] Roland T. Mittermeir and Andreas Bollin. Demand-driven Specification Partitioning. In *Proceedings of the 5th Joint Modular Languages Conference*, 2003.

[22] Hausi. A. Müller, Scott R. Tilley, and Kenny Wong. Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project. In *CASCON'93*, pages 217–226, October 1993.

[23] U. Nickel, J. Niere, J. Wadsack, and A. Zündorf. Roundtrip Engineering with FUJABA. In J. Ebert, B. Kullbach, and F. Lehner, editors, *Proceedings of the Second Workshop on Software-Reengineering (WSR)*, Bad Honnef, Germany, August 2000.

[24] Tomohiro Oda and Keijiri Araki. Specification slicing in a formal methods software development. In *Seventeenth Annual International Computer Software and Applications Conference*, IEEE Computer Socienty Press, pages 313–319, November 1993.

[25] Daniela Pohl. Specification Comprehension – Konzeptverwaltung am Beispiel zustandsbasierter Spezifikationen (in German). Master's thesis, University of Klagenfurt, Software Engineering and Soft Computing, Juli 2008.

[26] Daniela Pohl and Andreas Bollin. Database-Driven Concept Management – Lessons Learned from Using EJB Technologies. In *4th International Conference on Evaluation of Novel Approaches to Software Engineering*, May 2009.

[27] Denys Poshyvanyk and Andrian Marcus. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC2007)*, pages 37–48, June 26–29 2007.

[28] Václav Rajlich and Norman Wilde. The Role of Concepts in Program Comprehension. In *International Workshop on Program Comprehension*, pages 271–278. IEEE Computer Society, 2002.

[29] Rational-XDE. IBM Rational XDE DeveloperWorks Home Page. www.ibm.com/ developerworks/rational/products/xde/, Page last visited: March 2009.

[30] J.M Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 2^{nd} edition, 1992.

[31] Mark Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, 1979.

[32] T.A. Wiggerts. Using Clustering Algorithms in Legacy System Remodularization. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'97)*. IEEE Press, 1997.

[33] N. Wilde and M. C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7:49–62, 1995.

[34] Xinrong Xie, Denys Poshyvanyk, and Andrian Marcus. 3D Visualization for Concept Location in Source Code. In *Proceedings of 28th IEEE/ACM International Conference on Software Engineering (ICSE'06)*, pages 839–842, May 20–28 2006.

CHAPTER 10

DATABASE-DRIVEN CONCEPT MANAGEMENT – LESSONS LEARNED FROM USING EJB TECHNOLOGIES

D. Pohl, A. Bollin

Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering, pages 227–238. IEEE Computer Society, May 2009.

10.1 Abstract

During software maintenance activities one needs tools that assist in concept location and that provide fast access to already identified concepts. Thus, this paper presents an approach that is able to cope with this situation by storing concepts in a database. We demonstrate its applicability on formal Z specifications, where the huge number of concepts to be found emphasizes the use of an efficient database system. The paper closes with lessons learned, as the standard use of EJB-technologies redounds to more time-complexity than expected.

10.2 Introduction

As developers we are surrounded by complexity. Partly, this is because our applications get more sophisticated. Partly, this is because also our objectives get more and more complex. With it the related design documents explode in size and imply complications. This situation was already reflected by C.A.R. Hoare in the 1980s, who stated that *[..] there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no*

obvious deficiencies [11]. Small artifacts are rather exceptions, and locating deficiencies is a major business for a software personnel. The bad news is that fixing deficiencies is impeded by the very mentioned problems of size and complexity.

As scientists we are faced with the challenge to overcome at least parts of these hurdles. First, we have to sustain the understanding of relevant parts of a system and its related maintenance activities. Secondly, we have to ensure that the relevant parts (to be changed) can be located easily. These tasks are supported by software comprehension environments [18, 13, 3, 9], reverse engineering frameworks [17, 5, 8, 10, 14], and concept location tools [7, 24, 20]. They focus i.a. on either data-gathering, exploration and visualization of the code, and assist in knowledge organization. But concept location is not only restricted to programming languages. There are also techniques for formal specifications [1] or rule-based systems [23]. There, the approaches make use of the identification of relationships and the reconstruction of concepts by means of slicing, chunking, and clustering.

Despite these existing tools, concept location is still a laborious task. Over several periods of time often the same or similar concepts have to be reconstructed again and again, which is, additionally, a resource and time-consuming process. Therefore, this contribution suggests a framework that persistently stores conceptual elements and their dependencies in an SQL database. In 2008, a prototype has been implemented in the course of the master thesis of Pohl [19], and this contribution aims at sharing our experiences with the system and its evaluation.

This paper is structured as follows: Sec. 2 introduces the notion of concepts and their detection, first in general, and then in formal Z specifications. Sec. 3 presents the architecture of the framework and the related database model. Sec. 4 is dedicated to the use of Enterprise Java Beans (EJB). Sec. 5 describes the evaluation steps and lessons learned. Finally, Sec. 6 concludes this contribution with a short summary.

10.3 Concepts and Concept Location

When trying to understand a system, *concepts* are generally seen as *perceived regularities in events or objects, or records of events or objects, designated by a label* [21]. One is looking for related parts and trying to assign a name/meaning to them. Additionally, by aggregation, new (abstract) concepts can be built. The concepts we are looking for are exactly those parts with dependencies within and across artifacts.

Concept location is rather intuitive. Experienced users manage to navigate quickly around relevant parts but fail in explaining how they are excluding irrelevant parts. When their experience does not suffice, they follow three different strategies which are explained in more detail in [21]: (string) pattern matching, dynamic analysis, and static analysis. The process of concept location is iterative. By starting with a domain-level request, concept candidates are identified and evaluated with respect to their suitability and then they are either rejected or form the basis for the next evaluation step.

In order to demonstrate generality, we decided to focus on artifacts that are at a very high abstraction level and that are inherently complex: formal Z specifications [22]. They seem to be most useful as they are semantically very compact and are of a declarative nature. This implies that dependencies are definitely hard to identify, and the assumption is that other artifacts (like program text) will not complicate the situation.

Concept identification within formal specifications depends on the notion of control and data dependencies between their basic elements (also called *primes*). Their calculation is

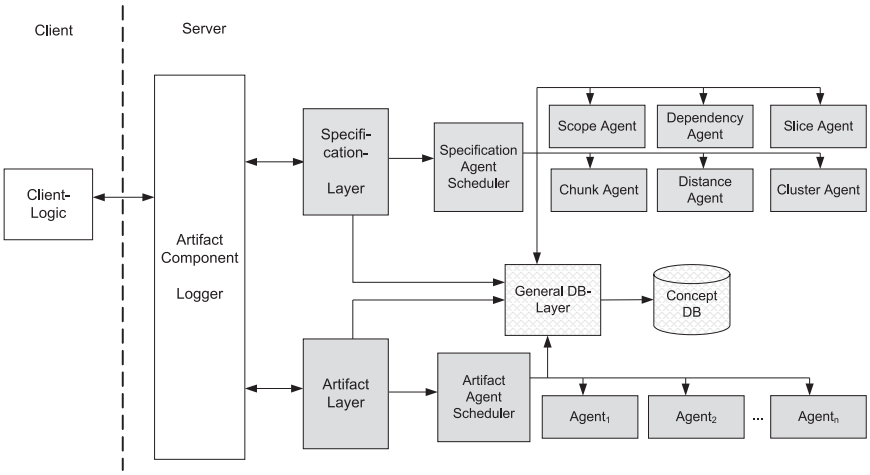


Figure 10.1 Architecture as implemented by the framework.

impeded by their declarative nature, but (with some limitations) they can be reconstructed. Basically, this is done by regarding scope rules, looking at the primes' identifiers, and, depending on their use, by assigning definition (D) declaration (T) and use (U) tags to them. Primes that describe an after state (they contain at least one D-tag) are said to be control dependent on primes that do not describe such an after state. When taking a specific identifier within the primes into account, data dependencies can be detected. For an in-depth discussion on dependencies and concepts see [2].

Based on the identified dependencies, the following partial specifications can be defined: specification slices, chunks, and clusters. Slices and chunks are generated by looking at a starting set of primes and by following control and/or data dependencies. Clusters are calculates by taking reachability considerations into account. These specification abstractions are hereinafter treated as concepts that are to be identified *via* and stored *in* the database. The following section introduces the architecture of the framework and the related database schema.

10.4 Architecture

The framework for identifying the different concepts in Z specifications is based on the architecture as shown in Fig. 10.1. It is designed to easily cope with different types of artifacts as the server is divided into three main parts:

1. **Artifact Independent Layer:** This layer represents the interface to the client and the component-logger.

2. **Artifact Depended Layer:** Depending on the type of the artifact (in our example Z specifications) this layer contains the control logic and the corresponding agents.

3. **General Database Layer:** This layer provides the necessary interface for manipulating the database.

The framework implements a traditional client-server architecture pattern. The client is responsible for visualizing the results and for triggering the concept extraction. On the

server side it is designed to handle different types of artifacts. The *Artifact-Component Logger* is responsible for the registration of different *Artifact-Layers*. Depending on the document to be stored (or accessed), the logger identifies the responsible layer. In our case (for a Z-Specification) the *Specification-Layer* is contacted. The artifact layers are responsible for implementing the necessary concept location functionality. Complex or time-consuming tasks (e.g. the dependency calculation) can be delegated to *Agents* that are synchronized by the *Artifact Agent Scheduler*.

Whenever a document is stored, different analysis tasks will have to be started to extract concepts, and the findings will have to be stored in the database again. In our prototypical implementation, every agent is responsible for one specific concept class. E.g. the *Dependency Agent* extracts data and control dependencies. The *Cluster Agent* is then used to calculate all possible clusters within one document. As clusters are created by looking at strongly-connected parts, this agent is scheduled not until the *Dependency Agent* has finished its calculation. A more detailed description of the agents is given in Sec. 10.4.2.

10.4.1 Database

The database schema (see Fig. 10.4 in the Appendix) can be divided into four parts: the *Management/Project Pane*, the *Representation Pane*, the *Concept Pane*, and the *View Pane*. Concept location can be treated as a multi-dimensional problem, and the dimensions, explained in more detail in [19], are just mapped to the corresponding parts in the schema. They are described hereinafter shortly:

During development and maintenance it is common to deal with different types and versions of documents, and the *Management/Project Pane* covers this functionality. Therewith, it is possible to store different *Artifacts* of the software engineering process. They are related to a specific *Project* and *Phase* (within they are generated).

An artifact consists of different *SyntaxElements*. As introduced above, in Z we call those elements primes. They form basic concepts of the artifact and are stored in the *(Syntactic) Representation Pane* of the database. The underlying structure of the document (and respectively of the elements) can be any simple or complex graph and is expressed by a circular n-to-m relationship. For future processing steps it is possible to annotate those elements. One type of annotation is the use of identifiers (e.g. T, D, or U) within primes (as mentioned earlier in Sec. 10.3).

Based on these annotations, concepts are extracted. They are handled within the *Model Concept Pane*. As there are different types of *Concepts* (e.g. dependencies, slices, cluster, and chunks) and as concepts can form hierarchies, again an n-to-m recursive relationship has been chosen to allow for greater flexibility. Additionally, for every concept, it is possible to store *ConceptMetaData* information.

Finally, different *Views* onto artifacts might exist. Those views cluster concepts of the same type together (e.g. all control dependencies of the document). Views are represented within the *(Concept) View Representation Pane*. The set of all views can be seen as a semantic snapshot of the whole document.

As mentioned above, every class of concepts is identified by an associated agent which is also responsible for storing the concepts in their corresponding views. The strength of the approach is its flexibility based on the use of a relational database. Agents just commit SQL queries to aggregate and store the necessary information. They are described in the following section.

10.4.2 Agents

Most of the work is done by agents. They are running independently from the client on the server side and interact with the database. It is possible to extend the framework by additional agents. The agents currently implemented in the prototype are: the *Scope Agent*, the *Dependency Agent*, the *Distance* and *Cluster Agent*, the *Slice Agent*, and the *Chunk Agent*.

Data and control dependencies between primes can only be detected when the scope (where the elements are involved) is clear. Thus, first the scope has to be extracted from the syntactical structure of the artifact.

$$\begin{aligned}\Pi_{sid}\ \sigma_{AnnotationType.name=\text{"}D\text{"}}\\ ((\sigma_{Concept.id=act}\,Concept \bowtie\\ (\sigma_{ConceptType.name=\text{"}State\text{"}}\,ConceptType))\\ \bowtie SyntaxElement \bowtie\\ ElementMetaData \bowtie AnnotationType)\end{aligned} \tag{10.1}$$

$$\begin{aligned}\Pi_{sid}(\sigma_{Concept.id=act}\,Concept \bowtie\\ (\sigma_{ConceptType.name=\text{"}State\text{"}}\,ConceptType))\\ [sid \neq sid]\\ \Pi_{sid}(\sigma_{\substack{AnnotationType.name\neq\text{"}T\text{"}\ or\\ AnnotationType.name\neq\text{"}C\ or\ AnnotationType.name\neq\text{"}D\text{"}}}\\ SyntaxElement \bowtie ElementMetaData\\ \bowtie AnnotationType)\end{aligned} \tag{10.2}$$

In our system the *Scope Agent* is responsible for that. In fact, what we informally call "scope" has three facets in Z: the *State Scope* which deals with schema and schema inclusions within a specification document, the *Connectivity Scope* which merges all primes of two or more schemata that are combined via schema operations, and the *Declaration Scope* that merges all primes that are necessary to keep it syntactically correct[1].

When the scope is fixed, the *Dependency Agent* is started and identifies control and data dependencies. Both, dependencies and scopes are stored as concepts within the database. An example of the *State Scope* can be found in Fig. 10.2 (out of the Birthday-Book specification of [22]). Due to schema inclusion, the primes of the "BB" state space are combined with the primes of the "Add" operation schema.

The next two agents are the *Distance* and *Cluster Agents*. For clustering formal Z specifications, distances between primes (across dependency paths) are taken. So the first agent calculates the distances between the primes in the specification. The second agent then calculates all potential clusters within one artifact.

Common abstractions with a clearly defined meaning are slices and chunks. The next two agents, the *Slice and Chunk Agent*, are responsible for extracting them. They look at

[1]The different types of scope are explained in more details in [19].

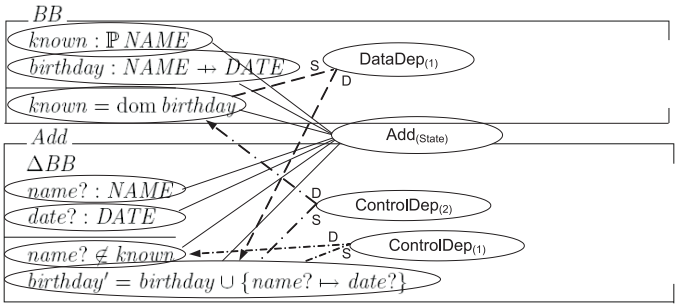


Figure 10.2 Scope and resulting data/control dependency.

every prime, take them as slicing/chunking criterion and, by following control and/or data dependencies, they calculate these forms of abstraction.

10.4.3 Dependency Agent

The *Dependency Agent* is described hereinafter in more details as its functionality demonstrates the ease of working with the database. Based on pre-identified scopes (that are already stored as concepts in the database), it is possible to calculate dependencies between syntactical elements.

To identify control dependencies within Z, some approximations can be conducted [1]: a syntactical element is control dependent upon another one, iff there is another element that decides whether the prime is evaluated or not. By utilizing the *use*-annotations (U) it is possible to identify these dependencies with ease. The calculation can be done by small queries (demonstrating the elegance of the approach). The queries (1) and (2) above calculate the start and the end positions of the control dependency arcs[2].

The dependency agent takes all results of the first query and connects them with the resulting elements of the second query. The same is done for data dependencies. Additionally, the identified pairs of dependencies are stored as concepts within the database. For the *Add-Operation* the resulting dependencies are shown in Fig. 10.2.

As the framework is assumed to be extend, maintainability was an important requirement during development. The choice dropped onto the EJB-Technology. The evaluation of the prototype also reflects on EJB internals, so the following section discusses the most important issues. More information about EJB can be found in [4].

10.5 EJB and Implementation Details

EJB 3.0 (Enterprise Java Beans) is a server-side middleware architecture of Sun Microsystems. The reason for choosing this technology was the non-functional requirement *maintenance* we wanted to guarantee, and EJB facilitates this separation between the application and the database logic. Additionally, it offers *bean-objects* to handle data easily and to map the relational data format to the object oriented paradigm respectively. This fact could be understood as an abstraction of the relational database schema in an object oriented presentation.

[2]The *act* identifier holds the scope for which the current calculation is to be performed.

Specification	Pages A4	Primes	CD	DD	ViZ(s)	EJB-A [s]	EJB-B [s]
BB	2	34	10	5	4.6	7.0	6.5
Cinema	4	74	121	43	75.3	43.2	30.7
Petrol	3	65	192	177	152.9	51.9	38.7
Elevator	6	185	1,628	992	1,223.4	709.3	502.7

Table 10.1 Complexity attributes and calculation time (in seconds) for experimental subjects.

The EJB technology is implemented via corresponding Java classes on the server which run in an EJB-Container. For the implementation of various functionalities different types of beans are provided. Special beans are needed for the connection to the database, the so-called *Entity Beans*. One object of an *Entity Bean* class holds one row of the appropriate table. Thus beans are the results of the object-oriented mapping provided by this technology.

The concept management framework is realized via *Java 1.6*, *EJB 3.0* and the Open-Source database system *MySql*. For the server side implementation the application server *Glassfish*[3] from Sun was used. The development environment was *NetBeans IDE 6.0.1*. The ORM (Object-Rational-Mapping) is provided by the *TopLink* persistence provider, developed by *Oracle*.

The framework implements the architecture as described in Sec. 10.4. The client is a stand-alone remote client and thus not executed in an EJB-Container. The server is implemented via EJB. The interface to the client (*Artifact Component Logger*) is represented by a stateless session bean. The different artifact dependent layers (as the *Specification Layer*) are also implemented as stateless session beans. Thereby, it is possible to serve more than one client at a time. This layer is also responsible to start the *Agent Scheduler*. The *Agent Scheduler* for Z specifications is a traditional Java class, which consults the agents as needed. The *Agents* are also traditional Java classes. For the persistency of the identified concept they get the entity manger from the current session. The interface to the database is formed by entity beans which map the database relations to the object oriented classes. Thus, these beans are contained as part in the *General DB-Layer*.

10.6 Evaluation

The evaluation of the framework was carried out in two steps. First, the correctness of the identified concepts were checked, and, secondly, the usefulness in respect to performance explored. In fact, both steps also hearken back to results of an existing framework called *ViZ* (for *Vi*sualization of formal *Z* specifications [2]). *ViZ* maps Z specifications to a graph (primes become vertices, dependencies are stored as arcs) and calculates dependencies based on reachability considerations.

[3]For further information about Glassfish see: https://glassfish.dev.java.net/, Last visit: Feb. 2009.

	incl. overhead in [s]	no overhead in [s]	diff (in %)
DD	139.2	93.3	-32.97
CD	343.6	232.5	-32.33

Table 10.2 Complexity, described by the number of data (DD) and control dependencies (CD).

Runs	JDBC [ms]	EJB [ms]	Factor
100	781	5,158	6.60
1000	7,784	51,767	6.65
10000	88,463	526,956	5.96

Table 10.3 Comparison of JDBC and EJB access to the database.

10.6.1 Setting and Correctness

The first step was the validation of the concepts that have been identified by the agents and stored in the database. The evaluation is based on wide-spread specifications of raising sizes, known as Birthday Book [22], Petrol Station [1], and Elevator [6]. Additionally, a student's specification (called Cinema) was added to the set, too. Tab. 10.1 (left side) presents the complexities of the specifications by exemplifying the number of pages (when pretty-printed), primes, control- (CD), and data dependencies (DD).

An in-depth description of the proof of correctness is out of the scope of this contribution. However, by exporting the results to a structured file it was possible to compare them with concepts described in literature and identified by the *ViZ* framework[4]. As every dependency and concept has been detected correctly, we were also eager to see whether the framework scales and improves operating speed.

10.6.2 Performance Considerations

The ViZ framework provides additional features (such as browsing the specification graphically), but the calculation of dependencies (and thereinafter slices or chunks) is time-consuming. Tab. 10.1 (right side) presents the time needed to calculate all dependencies, for the ViZ environment and the new framework (for two different settings, called EJB-A and EJB-B). For our approach we wanted to see whether there are some improvements or not. So, the performance[5] was explored thoroughly.

The reason for two settings was the inexplicable performance lack when working with specifications of raising sizes. The experiences we gained are described hereinafter. As most studies focus on the throughput of the system by varying the amount of clients served

[4]See [2] for more details on the meaning of specification clusters, slices, and chunks.
[5]We used the same measurement settings: *Intel(R) CP T2600 2.16GHz, 1GB RAM, Windows XP and ServicePack 2*

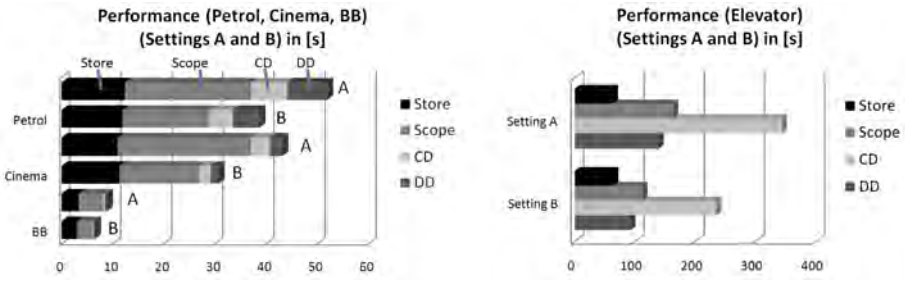


Figure 10.3 Time for storing primes, calculating scopes, and data and control dependencies.

by the EJB application [26, 15], this chapter approach the subject from a different angle (performance lacks due to database access of one client).

The performance of the system varied depending on the size of the specification, which was expected. Complexity considerations showed that the runtime complexity[6] is in $O(cs * 2n_s)$. Tab. 10.1 (right side) summarizes the time needed for the identification/storage of all primes and dependencies. The most complex artifact is the $Elevator$ specification[7], and after about 10 minutes it was analyzed and stored persistently for later use. On the same setting this is about two times faster than done by ViZ [1]. But we were eager to know why it took five minutes to store a bit more than 2600 data-sets.

We investigated further into this issue and made two important observations:

- Too much time is lost due to the EJB's synchronization between the database and Java's internal objects.

- There is very high execution time latency between EJB queries and their corresponding JDBC queries.

The measured times vary due to the different complexities of the specifications. But, performance is lost due to the overhead of the relational and object-oriented mapping. EJB can be seen as an additional layer between the DB and the implemented business logic. Every synchronization contributes to an increase in processing time. To get unique identifiers for objects (we used $auto\_increment\ ID$), one has to flush/synchronize the objects with those in the database. That this flush is costly was clear, but we wanted to know how much time is lost. We used the Elevator specification to measure it, and found out that the overhead is about one-third of the time (see Tab. 10.2).

The second issue we were curious about was the difference between EJB and JDBC when accessing the database. And indeed, we found a big time latency between EJB and JDBC queries. Not surprising, JDBC was faster, but the differences were notable (see Tab. 10.3). To measure it, we implemented the same requests with the EJB query language

[6]Here, cs is the number of different scopes, and n_s is the number of prime elements.
[7]There are 7,057 entries within the $combines$ relation of the database: 1,984 data dependency, 3,256 control dependency and 1,817 scope information (see Fig. 10.4).

Runs	JDBC [ms]	EJB [ms]	Factor
100	625	4,391	7.03
1000	6,315	43,485	6.89
10000	62,462	423,670	6.78

Table 10.4 Comparison of JDBC and EJB access to the database based on second settings.

	incl. overhead in [s]	no overhead in [s]	diff (in %) (in %)
DD	92.20	56.63	-38.58
CD	234.53	158.19	-32.55

Table 10.5 Time (concept manifestation w/o sync overhead) of setting two.

and with JDBC statements[8,9]. Then, both requests were issued up to 1000 times[10]. We found out that JDBC scales with the factor of about six times better than EJB.

So, although EJB (with entity beans and annotations within the entity bean classes) produces a more readable code, performance decreases when one has to store many objects per transaction which are, then, needed in ongoing processing steps. In our framework this is the case when we have to store an object and need the unique ID for storing the intermediate relation between those objects. The flushing/synchronizing mechanism is the only but very expensive way for getting it.

Additionally, the performance evaluation was accomplished with updated measurement characteristics[11]. This shows that upgrading the system is one and often the easiest way to achieve better performance results of EJB application. Tuning the operating system and platform is one factor suggested by Sun [16, p.95]. The evaluation showed that with improved CPU power and additional working memory the performance of EJB yields better results, as shown in Fig. 10.3 and Tab. 10.1 (settings A and B). However, independently from the setting, the speed-up when accessing the database (see Tab. 10.4) stays within the range of 5 to 7. Also the influence of the overhead remains constant (see Tab. 10.5) at about 30-40%.

An evaluation of an earlier draft of the EJB specification from Jordan [12] shows also significant performance differences between JDBC and EJB. As JDBC has to deal less with object oriented abstractions, it performs well with high database access rates. Another way to get higher performance is to de-normalize the database schema [25].

[8]The query tested for two equal identifiers at different primes and joined two times over the $SyntaxElement$, the $CombinationType$ entities, and the $combines$ and $emd\_annotates\_SE$ relation. The database contained 2620 entries in the $combines$ relation.

[9]This evaluation was performed with 150 entries within the $combines$ relation and 73 syntactical elements (see Fig. 10.4).

[10]Database internal optimizations, like cashes were, of course, disabled.

[11]Measurement settings: *Intel(R) Core(TM)2 CPU, T7200 @ 2.00GHz, 2 GB RAM*

Undoubtedly, EJB yields advantages like transaction management, security mechanisms, and scalability. It offers a comfortable way in implementing things. But this luxury does not come for free.

10.7 Conclusions

Concept location is a challenging task which also holds for the identification of concepts within formal Z specifications. Once detected, they should be stored for future use to save time when analyzing the artifacts again. For this reason a framework was implemented that is able to identify *and* store concepts in a database. For its implementation the middleware technology EJB was utilized.

This paper introduces the architecture and evaluates the resulting framework. The evaluation shows that it produces correct and useful results. However, the performance of the framework was strongly influenced by EJB. We found out that the most important latency is due to the synchronization process between a bean objects and the database. The comparison between JDBC and EJB shows a high factor of performance loss. JDBC scales about six times better than EJB in terms of runtime. Additionally, EJB implements an intermediate layer and, therefore, runs into performance latencies.

In a setting similar to our framework the evaluation shows that the use of EJB technologies is less suitable. EJB brings several maintenance advantages, but one has to expect a performance loss that should not be neglected.

REFERENCES

[1] Andreas Bollin. *Specification Comprehension Reducing the Complexity of Specifications*. PhD thesis, Institute for Informatics-Systems, University of Klagenfurt, 2004.

[2] Andreas Bollin. Concept Location in Formal Specifications. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(2):77–104, March/April 2008.

[3] Borland. The Rational Homepage. http://www.borland.com/us/products/together, 2008.

[4] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0*. O'Reilly, 2006.

[5] Ilene Burnstein, Katherine Roberson, Floyd Saner, Abdul Mirza, and Abdallah Tubaishat. A Role for Chunking and Fuzzy Reasoning in a Program Comprehension and Debugging Tool. In *TAI-97, 9^{th} International Conference on Tools with Artificial Intelligence*. IEEE press, November 1997.

[6] J. Chang and D. Richardson. Static and Dynamic Specification Slicing. In *In Proceedings of the Fourth Irvine Software Symposium, Irvine, CA*, April 1994.

[7] Kunrong Chen and Vaclav Rajlich. RIPPLES: Tool for Change in Legacy Software. In *IEEE International Conference on Software Maintenance*, page 230, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[8] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO – Generic Understanding of Programs An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[9] Eclipse. Generative Modeling Techn. Homepage. http://www.eclipse.org/gmt/, 2008.

[10] R. Ferenc, A. Beszedes, M. Tarkiainen, and T. Gyimothy. Columbus – Reverse Engineering Tool and Schema for C++. In *IEEE International Conference on Software Maintenance*, pages 172–181, Montreal, Canada, 2002.

[11] Charles Antony Richard Hoare. The emperor's old clothes. *Commun. ACM*, 24(2):75–83, 1981.

[12] Mick Jordan. A Comparative Study of Persistence Mechanisms for the Java Platform. In *http://research.sun.com/techrep/2004/smli_tr-2004-136.pdf*, Inc. 4150 Network Circle Santa Clara, CA 95054 U.S.A., 2004. Sun Microsystems Documentation.

[13] F. Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA 2005), Workshop on Traceability*, 2005.

[14] E. Korshunova, M. Petkovic, M. G. J. van den Brand, and M. R. Mousavi. CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code (Tool Paper). In *Working Conference on Reverse Engineering (WCRE'06)*, Benevento, Italy, 2006.

[15] Avraham Leff and James T. Rayfield. Improving Application Throughput With Enterprise JavaBeans Caching. *Distributed Computing Systems, International Conference on*, 0:244, 2003.

[16] Sun Microsystems. Sun Java System Application Server 9.1 Performance Tuning Guide. EJB Performance Tuning. In *http://docs.oracle.com/cd/E19159-01/819-3681/index.html*, Inc. 4150 Network Circle Santa Clara, CA 95054 U.S.A., 2007. Sun Microsystems Documentation.

[17] Hausi. A. Müller, Scott R. Tilley, and Kenny Wong. Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project. In *CASCON'93*, pages 217–226, October 1993.

[18] U. Nickel, J. Niere, J. Wadsack, and A. Zündorf. Roundtrip Engineering with FUJABA. In J. Ebert, B. Kullbach, and F. Lehner, editors, *Proceedings of the Second Workshop on Software-Reengineering (WSR)*, Bad Honnef, Germany, August 2000.

[19] Daniela Pohl. Specification Comprehension – Konzeptverwaltung am Beispiel zustandsbasierter Spezifikationen (in German). Master's thesis, University of Klagenfurt, Software Engineering and Soft Computing, Juli 2008.

[20] Denys Poshyvanyk and Andrian Marcus. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC2007)*, pages 37–48, June 26–29 2007.

[21] Václav Rajlich and Norman Wilde. The Role of Concepts in Program Comprehension. In *International Workshop on Program Comprehension*, pages 271–278. IEEE Computer Society, 2002.

[22] J.M. Spivey. *The Z Notation. C.A.R. Hoare Series*. Prentice Hall, 1989.

[23] Daniel Wakounig. *Reverse Engineering of Typed Rulebased Systems – Dependency Analysis and Comprehension Aspects*. PhD thesis, University of Klagenfurt, June 2008.

[24] Xinrong Xie, Denys Poshyvanyk, and Andrian Marcus. 3D Visualization for Concept Location in Source Code. In *Proceedings of 28th IEEE/ACM International Conference on Software Engineering (ICSE'06)*, pages 839–842, May 20–28 2006.

[25] S. S. Yao, R. Hiriart, I. Barg, P. Warner, and D. Gasson. A case Study of Applying Object-Relational Persistence in Astronomy Data Archiving. In P. Shopbell, M. Britton, and R. Ebert, editors, *Astronomical Data Analysis Software and Systems XIV*, volume 347 of *Astronomical Society of the Pacific Conference Series*, page 694ff, December 2005.

[26] Yan Zhang, Anna Liu, and Wei Qu. Comparing industry benchmarks for J2EE application server: IBM's trade2 vs Sun's ECperf. In *ACSC '03: Proceedings of the 26th Australasian computer science conference*, pages 199–206, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

APPENDIX

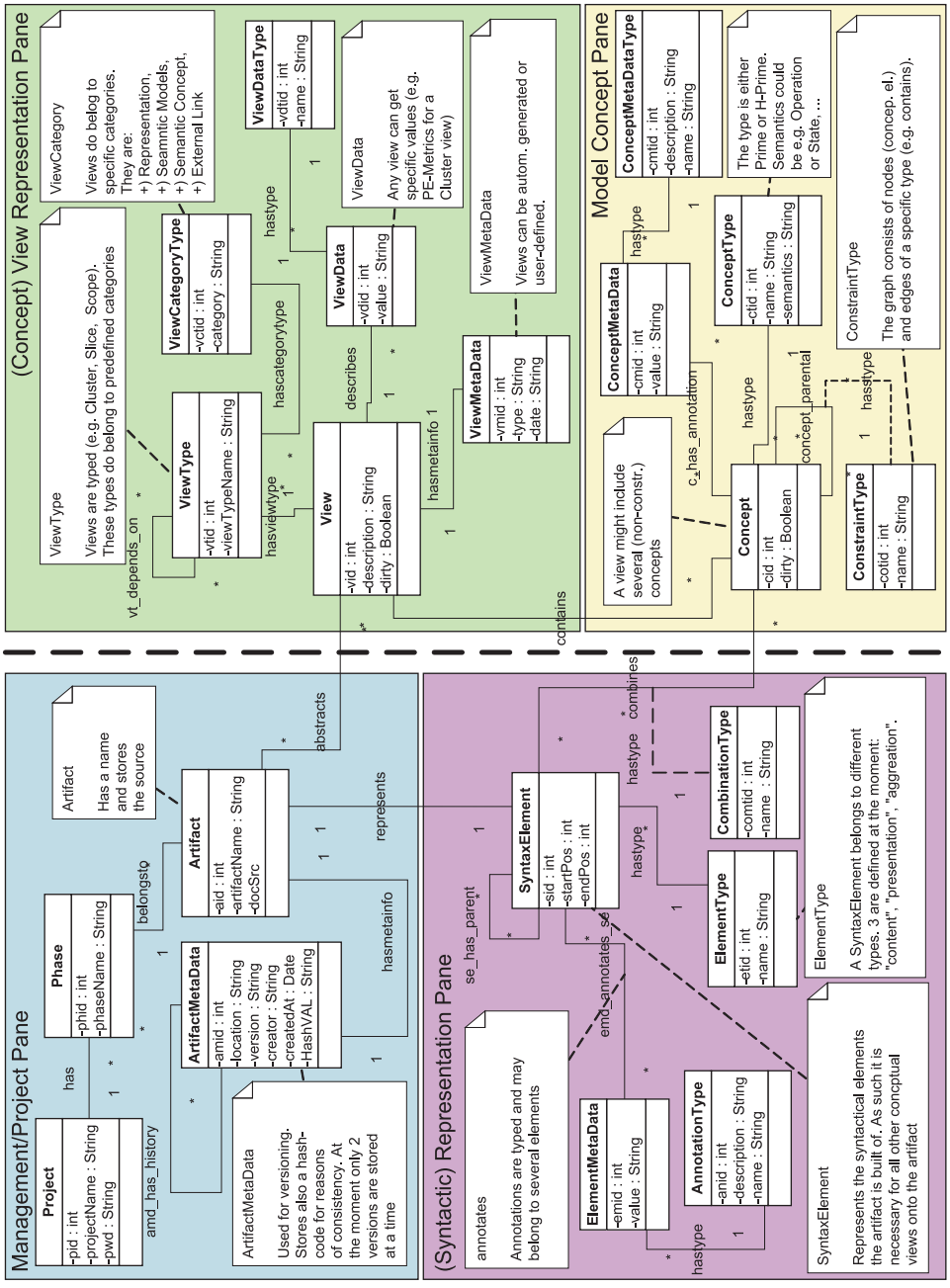


Figure 10.4 The four different panes of the database model.

PART V

FURTHER READING

CHAPTER 11

MAINTAINING FORMAL SPECIFICATIONS – DECOMPOSITION OF LARGE Z SPECIFICATIONS

A. Bollin

Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary, pages 442-453. IEEE Computer Society, 2005.

11.1 Abstract

Being part of different maintenance models formal specifications can act as valid artefacts for maintenance tasks. However, the linguistic density of specification languages and the size of specifications might still be seen as an obstacle against comprehension, reuse, and change activities.

This paper introduces an approach for the identification of specification fragments of Z specifications with a well defined semantic content. These fragments, namely specification chunks and specification slices, not only support comprehension tasks, they also enable maintenance personnel to identify and focus on the relevant parts of specifications for the problem at hand. Their ease in creation and use makes them well suited for maintenance, as is demonstrated by a simple prototype for Z specifications.

11.2 Introduction

Failure and success of formal methods depend on the viewpoint (and the person) from which (from whom) they are going to be assessed. There are studies and experience reports that tell about benefits in areas like railway and Metro systems, telecommunication, and

other security related fields of application [9, 19]. However, it often is argued that existing formal specification environments are not practicable at the moment [11], and a commonly known argument is: "Our program code exactly describes the behaviour of the system; why not throw away the specification/design documents and just take the underlying software code as the sole input?" – There are at least three aspects that are easily forgotten:

- Reconstructing all concepts from program code is a time-consuming task. It requires a lot of tool support which ultimately means the reconstruction of the missing documentation at different levels of abstraction.
- Specifications provide a trustful source for the description of the original requirements, especially when they are kept up to date during the evolutionary steps taking place in the course of system development.
- Even if specifications are getting large, they are smaller than the programs that are implementing the described requirements.

The benefit of a formal specification as a useful source for documentation, test, and refinement is true if and only if the specification is kept up to date during evolutionary steps taking place during development, and later, during operation and maintenance. To keep formal specifications up to date requires effort, and with growing sizes of specifications the situation even gets worse. The specification of the air traffic control system (CDIS, Central Control Function Display Information System [12]) had about 1000 pages, and managers often claimed that formal specification languages are merely "write-only" languages which do not scale up.

The problem is not new. Programs as well as specifications get complex if their size exceeds some limit. The good news is that for programs there are a number of approaches supporting maintenance tasks. The bad news is that there is no (tool) support for maintenance tasks concerning formal specifications. As maintainability of a system depends on the ease of maintaining the underlying system and documents, and as specifications are not that well supported, it is no wonder that formal specifications have not found their way into industrial practice.

This paper suggests an approach that eases comprehension and maintenance tasks of specifications. It enables the maintenance personnel to focus on those parts of the specification that are relevant for the problem at hand. Section 11.3 starts with an overview of the context of the maintenance tasks and discusses the state of the art. Section 11.4 presents the general idea of how to deal with complex formal specifications. Section 4 introduces a simple prototype for Z specifications implementing the presented approach. The paper concludes with a summary of recent findings and an outlook of how the approach is going to be improved.

11.3 Maintaining Specifications

The average lifetime of a software system is said to be about a decade. It thus is no wonder that more than half of the resources spent on the system are spent for maintenance activities [2]. These activities are not only influenced by the type of maintenance, they are highly influenced by the artefacts of the system available to the maintenance staff. These artefacts, their level of abstraction, and the underlying maintenance process model define the maintenance context.

11.3.1 The Maintenance Context

In [18] Pirker identified four contexts for software maintenance. His focus was on product composition, and depending on which artefacts are available at hand, he differs between classical systems and truly maintainable systems. When only the binaries are available this is referred to as the *pathological maintenance* context. When the source code is available, too, the situation is regarded to as the *classical maintenance* context. The *model-based maintenance* context already includes design and/or specification models. Finally, when additional clues (so called hooks) are available, the context is that of a *maintainable system*.

Of the four contexts the last two are most interesting for our considerations. Specification documents are available and support maintenance activities. However, they are used slightly differently depending on the underlying maintenance process model [1, 14]. Among them there are:

- The *Quick Fix Model* which focuses on code. Usually there is a change request on the code. After compilation and test this change influences all other artefacts, including the specification. In this situation it is important to *identify related parts within* the specification.
- The *Iterative Enhancement Model* might be compared to prototyping approaches. It starts with the analysis of the old system, which, in consequence, leads to a set of modified requirements. The whole life-cycle is repeated. Here, the specification is first used as an input source for *comprehension activities*, thus used to provide *concepts* to the maintenance personnel. Afterwards the specification is adapted to the changing requirements.
- The *Full Reuse Model* is based on the reuse of parts or components of the old system. The maintenance staff has to decide, which parts of the system are to be reused, and which are not. Here the specification, as an artefact, is analyzed for reuse, too. It is important to *carve out components* of the specification in order to decide whether they are suitable for reuse or not.
- The *Specification Based Maintenance Model* focuses on changing the specification and design documents before starting typical refinement steps. Here it is necessary to rapidly *identify dependent parts within* the specification (as they are likely to be influenced by a planned change, too).

Not only the models can be mixed up, the different activities of dealing with the specification are also interrelated. The requirements for an approach supporting specification maintenance are manifold. Maintenance support consists of support for

- analysis of change and ripple effects,
- design recovery, and
- restructuring the formal specification.

11.3.2 Impediments

Specifications may contain several thousand lines of specification text, and the deducted software system might consist of millions of lines of code. Due to the set of overwhelming

details the artefacts are said to be ”complex”. Up to this there is no difference between maintaining programs and maintaining specifications. In any situation the maintenance personnel has to fully comprehend the relevant parts of the system, and *size* is one of the most influencing (and limiting) criteria when dealing with the artefacts. But it is not only the size that matters.

As is argued by Mittermeir and Bollin in [16], it is apparent that people like to write code, but they do not like to read somebody else's code. This statement is not based on an empirical study but rests on experiences gained by talking to people and by observing students' as well as professionals' behaviour during software maintenance. Generally speaking, it is easier to express ones own concepts and ideas using the tight formality of a programming or specification language than to reconstruct the concepts the original developer had in mind.

Edmonds introduces the term *analytical complexity* [10] for this type of difficulty when trying to comprehend somebody's expressions. Besides size there are several reasons why the reconstruction of the original concepts behind a formal specification is that hard:

- Missing redundancy.
- Too few clues for reconstructing the original structure. Putting too much structure into a specification is usually understood to be a hint towards implementation.
- Too few clues for reconstructing the behaviour (due to the declarative nature of specifications).

Missing redundancy is a property of a formal specification but could partly be overcome by using rewrite-systems. However, rewrite-systems are time-consuming and require special skills. Structure and behaviour are also not easy to be recovered.

Here one strategy out of the field of program comprehension seems to be very promising: the generation of partial/reduced representations of the code. However, in general the underlying algorithms are based on the identification of control and/or data dependencies, and the bad news is that there are major differences between programs and specifications. In fact, due to the declarative nature of specifications there is no execution order and no direct flow of control – which makes typical program comprehension tools unusable. Nevertheless, the concept of partiality is worth looking at, and this section concludes with some related work in this field.

11.3.3 Related Work

The informatics literature contains several concepts of partiality, aiming to provide an interested party just the perspective needed for a particular task. Especially the notions of slices [21] and chunks [7] should be mentioned, and their applications can be found in various fields. Korel et.al. discuss the slicing of formal state machines in [13], and Zhao, e.g., introduces slicing of concurrent logic programming in [22] and slicing of software architectures in [23].

In 1993 Oda and Araki [17] first used static slicing techniques for analyzing Z specifications based on a simple definition of data-dependency. One year later, Chang and Richardson [8] introduced dynamic specification slicing (by extending the idea of Oda and Araki). A formal approach is that of Leminen [15], who defines slices based on logical and

precondition dependencies for the calculation of the cohesion of Z-schemata. However, no approach is based on a general notion of control and data dependencies, and thus the generation of other types of partial representations is impeded. With the formal definition of these dependencies in [4] this limitation should be abolished.

11.4 Maintenance Support

As argued in [16], the complexity of specifications is the main reason why developers shy away from using specifications. The density of expressing thoughts in specifications becomes detrimental for comprehending them during later phases. This section presents an approach of how to deal with the complexity of specifications and how these findings can be used in order to ease maintenance tasks.

11.4.1 Dealing with Complexity

Formal specifications remain compact structures, but the complexity of specifications can be reduced effectively. The key idea is to support the maintenance personnel with well-defined types of specification abstractions, namely *partial specifications*. The key ideas are:

- A partial specification is smaller than the original specification, but contains all relevant parts of interest (for the problem at hand).
- When partial specifications are substantially smaller than the full specification, they are easier to grasp. It is size that matters.
- In an optimal case partial specifications are derived from the full specification automatically.

11.4.2 Partiality

At the first glance various types of partial specifications (sometimes also called specification abstractions) are of interest. However, looking only at terminals (literals) of the specification language is not sufficient, and defining arbitrary sets of literals as units is not practicable either. Semantically meaningful units are needed. For the definition of suitable elements a bottom-up approach has been chosen:

11.4.2.1 Specification Literals In general, specifications are constructed from basic (atomic) units. These basic elements are called *specification literals*. They can easily be identified by looking at the grammar of the specification language. As an example, specification literals can be keywords of the specification language, any operators or identifiers. When looking at the Z set-comprehension expression

$$\{x : \mathbb{N} \mid x < 5\}$$

the set of specification literals is $\{\ '\{',\ 'x',\ '\ :',\ '\mathbb{N}',\ '\ |',\ '\ <',\ '5',\ '\}'\ \}$. However, specification literals are not very expressive when standing alone. It is the combination of literals that makes them rich in content.

11.4.2.2 Prime Objects By aggregating specification literals, *prime objects* of a specification can be built. In specification languages these prime objects can be expressions, predicates, or even generic type definitions or schema type definitions. Some examples of Z specification primes are the strings: "$Report ::= OK \mid NOK$" or "$[limit : \mathbb{N} \mid limit = 10]$".

Prime objects are not restricted to simple expressions. As they form logical units, the simple primes mentioned above can be combined together in order to form so-called higher-level primes.

Definition 33 Prime Object. *A specification prime object represents the basic entity of a specification it is built out of specification literals and forms logical, syntactic or semantic units.*

The following $Success$ operation schema in Z notation

$$
\begin{array}{|l}
\hline
Success \\
result! : Report \\
\hline
result! = OK \\
\hline
\end{array}
$$

is an example of such a higher-level prime. In literature the terms $modules$ and $operations$ are sometimes used to denote higher-level specification prime objects.

The important thing is that these prime objects are immutable in the sense that they form the fundamental units (states, operations) on which specifications are built upon. Moreover, it is also important to know that they are merely defined by syntactical rules of the specification language.

11.4.2.3 Specification Fragments The assembly of several specification prime objects leads to specification fragments.

A *specification fragment* consists of several prime objects, but does not necessarily constitute a complete specification. It is a composition of several primes which are isolated from their surrounding context. The following set of primes forms a simple specification fragment:

$$
\begin{array}{|l}
\hline
Add \\
\\
\hline
name? \notin known \\
known' = known \cup \\
\qquad \{name? \mapsto date?\} \\
\hline
\end{array}
$$

The Add fragment is an incomplete portion of specification code. It consists of two primes checking and modifying the state. The exact meaning of Add becomes clear when realizing that $name$ is a state variable which contains pairs of names and dates of birth. A surrounding text is necessary.

The above fragment is not a higher-level prime object as it does not form a semantic unit in the specification (in our case at least a complete schema operation). This is a

key property of specification fragments: it is an incomplete or isolated portion of (specification) code that cannot be understood without a surrounding context, an explanation or commentary.

Literals, primes, and fragments are *basic elements* of a specification. However, to sustain comprehension tasks higher-level specification concepts are also needed – concepts that bear specific types of semantics. Here several types of abstractions can be derived: chunks, slices and clichés (which are discusses in more detail in [4]) form so-called *semantic elements* of a specification.

11.4.2.4 Specification Chunks *Chunks* are syntactic or semantic abstractions of text structures. In accordance with the definitions provided in [7], a specification chunk is a specification fragment that achieves a coherent purpose and can be understood outside of the context in which it is used.

Definition 34 *A* **specification chunk** *is*
(i) a prime including all primes contained within it, or,
(ii) a set of primes that exists within the same specification scope. For each pair of primes within the set of primes either one prime is dependent on the other or both primes are dependent on a third prime (within the set of primes).

The really important thing is that a chunk always has to be comprehensible in isolation. Compared to a specification fragment, a chunk contains enough (surrounding) context to stay understandable. That means that at least enough semantic information has to be taken into consideration when generating specification chunks.

The following Z specification is derived from a popular example in the formal methods literature [20]. It is the specification of a simple database called birthday book (BB, see App. A for the full specification) written in its horizontal form. It allows storing, searching for and deleting names and dates of birth.

$$
\begin{array}{l}
[NAME, DATE] \\
Report ::= OK \mid NOK \\
BB == [known : \mathbb{P}\, NAME;\ birthday : NAME \pfun DATE \mid \\
\qquad known = \mathrm{dom}\, birthday] \\
InitBB == [BB \mid known = \varnothing] \\
Add == [\Delta BB;\ name? : NAME;\ date? : DATE \mid \\
\qquad name? \notin known; \\
\qquad birthday' = birthday \cup \{name? \mapsto date?\}] \\
Delete == [\Delta BB;\ name? : NAME \mid name? \in known; \\
\qquad birthday' = birthday \setminus \{name? \mapsto birthday(name?)\}] \\
Find == [\Xi BB;\ name? : NAME;\ date! : DATE \mid \\
\qquad name? \in known;\ date! = birthday(name?)] \\
Success == [report! : REPORT \mid report! = OK] \\
FunctioningDB == Add \land Delete
\end{array}
$$

Based on the above birthday book specification, a specification chunk can be generated by looking at the specification prime $birthday' = birthday \cup \{name? \mapsto date?\}$ in the Add operation schema and by regarding all primes which are data dependent on that prime and which are necessary to ensure the correct syntactical context:

$$
\begin{array}{l}
[NAME, DATE] \\
BB == [known : \mathbb{P}\, NAME;\ birthday : NAME \pfun DATE \mid \\
\quad known = \mathrm{dom}\, birthday] \\
Add == [\Delta BB;\ name? : NAME;\ date? : DATE \mid \\
\quad birthday' = birthday \cup \{name? \mapsto date?\}] \\
Delete == [\Delta BB;\ name? : NAME \mid \\
\quad birthday' = birthday \setminus \{name? \mapsto birthday(name?)\}]
\end{array}
$$

The previous chunk describes how entries are added to and deleted from the database. It is substantially smaller than the original specification. The initialization schema, two operations ($Find, Success$) and several primes (e.g. $name \notin known$) have been omitted. Nevertheless, the chunk is understandable. The reason for the inclusion of the two primes containing the definition of $birthday'$ is that there is data dependency between them. The identification of such dependencies is not a trivial task and will be discussed in more detail in section 11.4.3.

11.4.2.5 Specification Slices The approaches presented in section 11.3.3 generate their slice in a top down manner, by removing elements from the specification.

Beginning with a large, but formally correct specification, it is advisable to start the slicing process in a *bottom up manner* – at the slicing criterion which should be at least a prime object. This assures that each slice which is carved out of a specification has well defined semantics. And the slice is driven by this prime's semantics. By aggregating additional primes properly, the resulting specification fragment will always have defined semantics.

Proceeding the other way round (in a top down manner) would make the process much more difficult. A function that deletes "the parts not needed" then has to be applied to the specification. Either the word "needed" gets very complex semantics, or the semantics of the fragmental specification cannot necessarily be given.

Definition 35 *A* **slicing criterion** *of a specification determines a specific point of interest in the specification. It consists of a specification prime and a set of literals which are element of the specification prime.*

Definition 36 *A* **specification slice** *is a syntactically and semantically correct specification which is the result of adding those primes to an (initially empty) specification which are directly or indirectly contributing to the slicing criterion.*

In contrast to the concept of a specification fragment (and the definition of a specification chunk), Def. 36 demands that a *specification slice* is both syntactically AND semantically correct. With respect to the slicing criterion "$birthday' = birthday \cup \{name? \mapsto date?\}$" in the Add operation schema the slice leads to the following BB-specification slice:

$$
\begin{array}{l}
[NAME, DATE] \\
BB == [known : \mathbb{P}\, NAME;\ birthday : NAME \pfun DATE \mid \\
InitBB == [BB \mid known = \varnothing] \\
Add == [\Delta BB;\ name? : NAME;\ date? : DATE \mid \\
\quad name? \notin known; \\
\quad birthday' = birthday \cup \{name? \mapsto date?\}] \\
Delete == [\Delta BB;\ name? : NAME \mid \\
\quad name? \in known; \\
\quad birthday' = birthday \setminus \{name? \mapsto birthday(name?)\}] \\
FunctioningDB == Add \land Delete
\end{array}
$$

The previous slice represents a syntactically correct specification. It contains all primes that are directly and indirectly contributing to the slicing criterion. But it is smaller than the original specification. In fact, two operation schemata ($Success$ and $Find$) are omitted. $Success$ does not contribute to the application of the prime and $Find$ does not modify or influence the value of the $birthday$ state variable.

11.4.3 Hidden Dependencies

As mentioned above, declarative specification languages do not provide an explicit notion of control. On the other side, the definition of control is based on the concept, that a statement is evaluated. This evaluation then decides whether another statement is executed or not. A similar concept can be identified within specifications: pre- and post-conditions. The pre-condition part is evaluated and this evaluation determines whether the post-condition part of the specification is applicable or not [4].

The following Add schema of the birthday book specification contains two predicates. The first predicate checks whether the provided name is in the birthday database. The second predicate contains an after-state identifier ($birthday'$) and adds the name and the accompanying date to the database of birthdays. These two predicates can be interpreted as a *pre-condition prime* and a *post-condition prime*.

$$\begin{aligned} Add == [&\Delta BB;\ name? : NAME;\ date? : DATE \mid \\ &name? \notin known; \\ &birthday' = birthday \cup \{name? \mapsto date?\}] \end{aligned}$$

The crux of the matter is that a semantic analysis of the pre and post-conditions does not generally lead to the above primes. The situation gets worse when Z schemata are combined by using Z schema operators like composition ($\fatsemi$) or implication ($\Rightarrow$). In [4] Bollin introduced the notion of a syntactical approximation to the semantic analysis and ended up with a simple set of rules for the identification of relevant prime candidates. They are summarized in Tab. 11.1. E.g. when two schemata S and T are combined by using sequential composition ($\fatsemi$), then there is control dependency ($\rightrightarrows_c$) between post-condition primes in S (po_S) and T (po_T) and the pre-condition prime of S (pr_S).

The following definition for the identification of control dependencies within schema boxes can be advanced:

Definition 37 Control dependencies in Z schemata. *Let S be a schema of a syntactically correct specification. Furthermore, let pr_S be the non-empty set of pre-condition primes of S and po_S the non-empty set of post-condition primes of S.*

Then primes in po_S are said to be control dependent on primes in pr_S (abbreviated as $po_S \rightrightarrows_c pr_S$).

When Z schemata are combined using Z-schema operators, then primes in po_S are said to be control dependent on primes in pr_S in respect to the rules presented in Table 11.1.

With the notion of control, the definition of data dependency gets possible, too. The parts to look for are the definition (or assignment) of values and the use of data elements. In Z, a literal denoting a data element is said to be an identifier. According to the terminology used in the Z community, an identifier is said to be *declared*, if it appears at the left side

Schema Operation	Related Primes
S	$po_S \rightrightarrows_c pr_S$
$\neg\, S$	$po_S \rightrightarrows_c pr_S$
$S \lor T$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \Rightarrow T$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \land T$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \Leftrightarrow T$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \upharpoonright T$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \fatsemi T$	$(po_S \cup po_T) \rightrightarrows_c pr_S$
$S \gg T$	$(po_S \cup po_T) \rightrightarrows_c pr_S$

Table 11.1 Control dependency calculation differs, depending on the type of schema operation.

of a declaration or at the left side of a schema expression. It is said to be *defined*, if the identifier is decorated and appears at the left side of a value assignment. It is said to be *used*, if it is neither declared nor defined.

Based on this terminology, data-dependency in Z specifications is defined as follows:

Definition 38 Data dependency between Z-primes. *A specification prime p is* data dependent *on a specification prime q $(p \neq q)$ if*
(i) there exists at least one identifier v (literal denoting a data element) that occurs in both p and q, and
(ii) v is defined in q and used in p, and
(iii) either p and q are in the same scope, or p is control dependent on q.

With the introduction of the notion of control in Z specifications, it gets possible to define control and data dependencies within state-based specifications. However, it is just a means to an end. Up to now several forms of abstractions have been discussed. Not all abstractions are useful in all situations. Their applicability depends on the problem at hand and thus on the point of interest.

11.4.4 Applicability

Section 2.1 already discussed the different situations in which formal specifications might be involved during maintenance tasks. The previous section introduced the notions of prime objects and dependencies, and in the sequel the applicability of the approach is discussed. It refers to the requirements for tool supporting specification maintenance.

- *Analysis of change and ripple effects*. A requirement changes and thus a small part (e.g. a prime) of the specification is about to change, too. It gets necessary to see the minimal portion of the specification affected by the change. As slices guarantee the inclusion of all dependent primes, this situation can be controlled by generating *specification slices*.

- *Design recovery*. In order to understand the specified system, it gets necessary to focus on minimal portions of the specification text. Not all dependencies are of interest at the same time and locality is usually more important than global relationships. This situation can be controlled by generating specification *chunks*. On the other hand both specific and distributed portions (fragments) of the specification text might be of interest. This situation could arise when looking for operations that modify a well defined set of state variables.

- *Restructuring*. Again, slicing and chunking techniques can be applied when identifying and changing whole portions of the specification text. A chunk (or set of chunks) helps focusing on specific parts of the specification, and, whenever the focus is clear, the slice guarantees that all relevant portions of specification text can be considered.

The above mentioned maintenance tasks can all be sustained by slices, chunks, and related fragments. This does not mean that other approaches are to be excluded, but with partial specifications two important activities become possible: focusing on *small parts* of the specification and putting an eye on *relevant* (which means *dependent*) parts of the specification. With slices, chunks, and fragments, the basis for useful abstraction is provided.

What remains is the question of how to focus on the point of interest. As with program slicing it is the abstractions criterion that influences the result, and in analogy to a program's slicing criterion, the criterion for specifications consists of several parts:

- Firstly, the focus will have to be set to a specific position in the specification. As argued in section 2.1 it only makes sense to look at the smallest entity with a well defined semantics available in a specification: the specification *prime*.

- Secondly, different types of abstraction require an adjustable focus. The focus itself can be set by making use of two features: in the first place there are the types of *dependencies* that are of interest. However, adjusting the focus via inclusion or exclusion of dependent primes is a rather coarse mechanism. Thus, the focus should additionally be adjusted by considering specification *literals*.

Thus the criterion for creating specification abstractions consists of three parts: a specification *prime* (representing the point of interest), a description of relevant *dependencies* (that have to be considered) and a set of *literals* (for optionally fine-tuning the selected dependencies).

Based on the definitions of primes and dependencies, slices and chunks can be generated automatically from a given specification. All that is necessary is to provide the appropriate abstraction criterion. Section 4 introduces a simple prototype that is able to support the above mentioned maintenance activities for Z specifications.

11.5 A Prototype for Maintenance Support

In order to deal with large specifications, a simple text-based prototype for Z has been implemented. It serves as a basis for the experiment presented in the remainder of this section. This section describes the prototype in more detail.

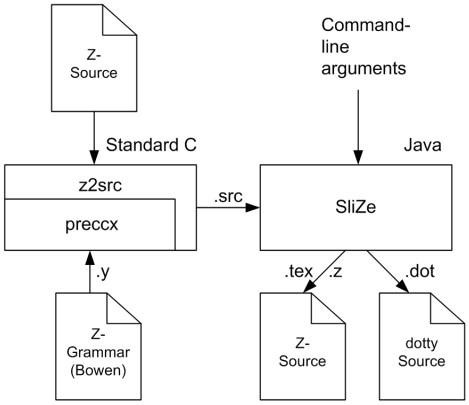


Figure 11.1 General structure of the prototype.

11.5.1 Technical Background

The following tasks can be handled by the prototype:

- It identifies dependencies hidden in the specification.
- It calculates partial specifications, that is chunks and specification slices.
- It transforms the specification to an augmented net (in analogy to a dependence graph) and visualizes the specification such that primes are represented as vertices and dependencies are represented as arcs.
- It calculates statistics based on the net representation of the specification. This includes the number of primes, the number of control-dependencies, and the number of data-dependencies.
- It generates output in two ways. Firstly, the net can be stored in a graphical format in order to visualize the net via *dotty*. Secondly, the net can be transformed backward to the specification source.

To ensure portability, the prototype (in the following called "*SliZe*" toolkit) has been implemented in Java. It is available for Windows and Linux platforms. For reasons of simplicity, the *SliZe* toolkit is based on the *preccx* grammar[1] of Z which has been defined by Breuer and Bowen [6]. The compiler produced by *preccx* is able to check for syntactically correct Z specifications written in LaTeX and has been modified in order to produce an intermediate representation of the specification. This representation then serves as an input to the *SliZe* application (see Fig. 11.1).

The prototype has some limitations when displaying and interacting with the net. As dotty tries to optimize the layout of the net, large specifications lead to very high computation times. A solution would be to abstract from the details in the net and to provide some sort of fish-eye views. Another limitation is that the prototype only provides a text-based interface.

[1]Breuer and Bowen's *preccx* home-page: http://www.afm.lsbu.ac.uk/archive/redo/precc.html. Page last visited: March 2004.

Name	#P	V	A	CC	v'(l)	v'(u)	DU
BB	1	72	267	24	4	10	4
Petrol	3	134	674	53	10	28	131
Elevator	6	349	3668	144	32	1096	1212
WM	12	520	2644	213	39	544	215

Table 11.2 Complexity overview regarding four Z-specifications.

Despite these limitations, the prototype proves to be useful. It simplifies the analysis especially when generating different partial specifications. In the context of an experiment more then 600 partial specifications have been generated and thousands of dependencies have been detected. And all can be done within a few minutes.

11.5.2 Experiment

In order to compare the specifications in respect to their complexity, the following measures are introduced [4]:

- Conceptual Complexity $CC(\Psi)$ of a specification Ψ. This measure is based on the number of basic prime objects in the specification (and not on the imprecise number of lines of specification code).
- Extended Cyclomatic Complexity $v'(\Psi)$ of a specification Ψ. Based on the number of control dependencies, lower and upper bounds for decisions in a specification are counted.
- The DU count metric $DU(\Psi)$ of a specification Ψ. By counting the number of data dependencies the maximum number of data relationships is identified.

With the additional definition of the number of vertices and arcs in the net presentation, a wide range of specifications' complexity measures exists. Comparisons between different specifications become feasible.

The birthday book specification is too small to benefit from the suggested approach. For that reason three other specifications have been looked at. The "Elevator" specification [8], the "Petrol-Station" specification ($Petrol$ for short) which was used during class labs at the University of Klagenfurt and the "ITC Window Manager"-specification (WM for short) which is a commercial specification presented in [5]. Table 11.2 summarizes the complexity of the specifications and the complexity of the net representation.

As can be seen, the BB-specification is the simplest one. The $Petrol$-specification is also small, but contains twice as much primes (and about twice as many vertices and arcs). The BB-specification consists of 72 vertices; the $Petrol$-specification consists of 134 vertices. The same ratio holds for the other measures, except for the DU count metric. Here, the $Petrol$-specification contains 131 data-dependencies; the BB-specification contains only 4 data-dependencies.

The specifications have been used to investigate the question whether and to what extent the generation of partial specifications pays off. The BB, Petrol, Elevator and WM

Specification	*Slice*	$Chunk_1$	$Chunk_2$	Mean
BB	0.80	0.73	0.81	0.78
Petrol	0.83	0.73	0.48	0.68
Elevator	0.86	0.33	0.22	0.47
WM	0.64	0.32	0.32	0.43

Table 11.3 Reduction factor k ($= Size_{new}/Size_{old}$) for four different specifications and the generation of slices and two different types of chunks.

specifications were used as experimental objects for the treatment: the application of the generation of partial specifications. The prototype mentioned in section 4.1 has been used to generate all possible sets of partial specifications for every predicate prime in the specification. These primes acted as the points of interest[2].

For every point of interest three different types of partial specifications were generated. For every prime vertex a full static specification slice, a full static specification chunk focusing on data-dependency, and a full static specification chunk focusing control dependency were calculated.

The experiment, particularly the efficiency of the generated partial specifications, is discussed in more detail in [3]. The subsequent section summarizes the two most important findings in respect to maintenance tasks.

11.5.3 Results

By automating the generation of slices and chunks it is possible to reduce the time for detecting the minimal portion of the specification to be changed to a few seconds. Once the specification is analyzed by the *sliZe* toolkit it is not necessary to look for hidden control and data dependencies within the specification text by hand anymore. Especially the generation of chunks leads to a drastic reduction of size, and with it to a drastic reduction of time needed to comprehend the partial specification. In respect to size complexity the prototype definitely prove useful.

The experiment demonstrated that the effect obtainable by slicing and chunking rises with the size of the specification under consideration. For the mean values of the above introduces complexity measures this observation had been confirmed (see table 11.3). Furthermore it could be stated that the slicing/chunking approach decreases complexity to a much greater extent when specifications are getting larger. It was also shown that the mean value of the increase of complexity of the generated partial specification is definitely less than the increase of the size of the specification.

These findings encourage the use of the approach for specification maintenance activities.

[2]In the BB- specification there are 7 points of interest. The Petrol- specification contains 21 points of interest, the Elevator- specification contains 102 points of interest, and the WM- specification contains 92 points of interest.

11.6 Conclusion

Specifications are valid sources in different maintenance process models and lead to an increase in maintainability. However, when there are no built-in hooks complexity hinders typical maintenance and comprehension tasks. This paper discusses typical impediments, but also elaborates on the different situations when specifications prove useful.

Formal specifications remain compact structures. However, in respect to maintenance and comprehension tasks the complexity of specifications can be reduced effectively. This can be achieved by focusing on those parts which are necessary to solve a specific problem at hand. Several factors contribute to the overall complexity, but the vast majority of problems goes back to the complexity of size; thus this work presents an approach to reduce the size of specifications. It is suggested to generate well-defined partial specifications such as specification slices and specification chunks.

The paper also introduces a prototype that has been written to sustain the maintenance of Z specification. The prototype will have to be improved to better incorporate into existing working environments – especially the lack of a graphical user interface has to be eliminated. However, the small experiment shows that the prototype is useful and that the generation of partial specifications does make sense in respect to maintaining formal specifications.

REFERENCES

[1] R. V. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Software*, 7(1):19–25, 1990.

[2] Keith H. Bennet. *Software Maintenance: A Tutorial. In M. Dorfman and R. H. Thayer, Software Engineering*, pages 289–303. IEEE Computer Society Press, 1997.

[3] Andreas Bollin. The efficiency of specification fragments. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering*, 2004.

[4] Andreas Bollin. *Specification Comprehension – Reducing the Complexity of Specifications*. PhD thesis, University of Klagenfurt, 2004.

[5] Jonathan Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press (ITCP), 1996.

[6] Peter T. Breuer and Jonathan P. Bowen. A concrete Z grammar. Technical Report PRG-TR-22-95, Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK, 1995.

[7] Ilene Burnstein, Katherine Roberson, Floyd Saner, Abdul Mirza, and Abdallah Tubaishat. A role for chunking and fuzzy reasoning in a program comprehension and debugging tool. In *TAI-97, 9th International Conference on Tools with Artificial Intelligence*. IEEE press, November 1997.

[8] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[9] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions, CMU computer science technical report CMU-CS-96-178. Technical report, Carnegie Mellon University, August 1996.

[10] Bruce Edmonds. *Syntactic Measures of Complexity*. PhD thesis, University of Manchester, 1999.

[11] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2003.

[12] J. Anthony Hall. Using formal methods to develop an atc information system. *IEEE Software*, pages 66–76, March 1996.

[13] Bogdan Korel, Inderdeep Singh, Luay Tahat, and Boris Vaysburg. Slicing of state-based models. In *Proceedings of the International Conference on Software Maintenance (ICSM'03)*. IEEE Press, 2004.

[14] Kevin Lano and Howard Haughton. A specification-based approach to maintenance. *Journal of Software Maintenance: Research and Practice*, 3:193–213, 1991.

[15] Janne A. Leminen. Slicing and slice based measures for the assessment of functional cohesion of z operation schemas. Master's thesis, Department of Computer Science, Michigan Technological University, 1994.

[16] Roland T. Mittermeir and Andreas Bollin. Demand-driven specification partitioning. In *Proceedings of the 5th Joint Modular Languages Conference, JMLC'03*, 2003.

[17] Tomohiro Oda and Keijiri Araki. Specification slicing in a formal methods software development. In *Seventeenth Annual International Computer Software and Applications Conference*, IEEE Computer Socienty Press, pages 313–319, November 1993.

[18] Helfried Pirker. *Specification based Software Maintenance (a Motivation for Service Channels)*. PhD thesis, University of Klagenfurt, 2001.

[19] Ann E. Kelley Sobel and Michael R. Clarkson. Formal methods application: An empirical tale of software development. *IEEE Transaction on Software Engineering*, 28(3):308–320, March 2002.

[20] J.M. Spivey. *The Z Notation*. C.A.R. Hoare Series. Prentice Hall, 1989.

[21] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.

[22] Jianjun Zhao. *Program Dependence Analysis of Concurrent Logic Programs and Its Application*. PhD thesis, Kyushu University, December 1996.

[23] Jianjun Zhao. Extracting reusable software architectures: A slicing-based approach. In *ESEC/FSE'99 Workshop on Object-Oriented Reengineering Toulouse (France)*, September 1999.

Appendix A - Birthday Book Specification

The birthday book (BB for short) describes a simple system for administrating names and birthday dates.

First names and dates are introduced as global sets. In order to indicate the success or failure of an operation, a global type $REPORT$ is introduced.

$$[NAME, DATE]$$
$$REPORT ::= OK \mid NOK$$

The state space consists of the set of all known names, and the "database" entries for the birthday dates. The predicate ensures that only known names are in the database.

BB

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$

$$known = \mathrm{dom}\, birthday$$

At the beginning the database is empty.

InitBB

$$BB$$

$$known = \varnothing$$

There are several operations for working with the database. It is possible to Add a pair $(name, date)$ to the database, it is possible to $Delete$ an entry from the database, and to $Find$ a birthday date in the database.

Add

$$\Delta BB$$
$$name? : NAME$$
$$date? : DATE$$

$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

Delete

$$\Delta BB$$
$$name? : NAME$$

$$name? \in known$$
$$birthday' = birthday \setminus \{name? \mapsto birthday(name?)\}$$

Find

$$\Xi BB$$
$$name? : NAME$$
$$date! : DATE$$

$$name? \in known$$
$$date! = birthday(name?)$$

To indicate the success of an operation the result OK is returned.

Success

$$result! : REPORT$$

$$result! = OK$$

With the above operation schemata the functioning system consists of successfully performed add or delete operations.

$$FunctioningDB == \\ (Add \land Success) \lor (Delete \land Success)$$

CHAPTER 12

THE EFFICIENCY OF SPECIFICATION FRAGMENTS

A. Bollin

Proceedings of the 11th Working Conference on Reverse Engineering, Delft, The Netherlands, pages 266–275. IEEE Computer Society, 2004.

Abstract

Formal specifications are valid sources for comprehension tasks when used during later development phases. However, the linguistic density of specification languages and the size of specifications can still be seen as an obstacle against comprehension activities.

This paper presents an approach for the identification of fragments of Z specifications with a well defined semantic content. These fragments, specification chunks and specification slices, are analyzed in respect to their efficiency when used during typical comprehension tasks.

12.1 Introduction

Formal methods and the application of formal specification languages play a crucial role in software engineering. During initial phases they are recommended as means to produce high quality software. However, using formal specifications at early steps exclusively is unfavorable in so far, as they can also be used as important drivers for test data generation [1]. When specifications are kept up to date during the various evolutionary steps of system

development they can also play a vital role during reverse engineering activities and the identification of concepts [13].

It is not without effort to keep specifications constantly up to date. The effort only pays off when additional benefits can be obtained. Such a benefit can be achieved by speeding up specification comprehension.

It has already been argued in [11] that the density of expressing thoughts is a positive attribute during development (from the perspective of the specification's writer). But this density becomes detrimental for specification comprehension during later phases. This is even worse when specifications are getting really large and the bulk of information exceeds dozens (if not hundreds) of pages of specification code. It is this observation of compactness that confirms the myth that formal specifications do not scale up.

The inherent density of specification languages cannot be changed. But comprehension problems aggregate when problem-inherent complexity is combined with the complexity of size. The latter can be reduced if a formal mechanism is devised which ensures that the user of a specification is presented only the portion of the specification (in the latter called specification fragment) relevant to the particular problem at hand [4].

In code comprehension, slices [17] and chunks [6] have been proposed as mechanisms for the identification of well defined portions of code. Both approaches guarantee that all that needs to be studied for the problem at hand is presented to the user. The notion of a specification slice has been introduced in [12], however, several types of specification fragments (to which specification slices and chunks are belonging to) are formally defined in [2] for the first time.

It is a legitimate question whether the approach of generating specification fragments can play the same role during reverse engineering steps as slices and chunks do. However, the usefulness depends on the effects (and limits) gained by the approach, and it has to be clear what can be expected when generating specification fragments.

As a first step this work focuses on the complexity of specifications – and with it on the complexity of the underlying comprehension task. The evaluation presented in the second part of this paper demonstrates that, especially with larger specifications, the generation of specifications fragments is very efficient. Complexity and size can be reduced in a practicable manner, which is a strong evidence for the usefulness of the approach.

The paper is organized as follows: the subsequent section introduces different types of specification fragments and discusses the detection of dependencies necessary for the generation of slices and chunks. Then the complexity of specifications is discussed and a basis for the calculation of complexity measures is provided. The paper then examines the efficiency of the approach of generating specification fragments, and it closes with a discussion of the results of the experimental studies.

12.2 Specification Fragments

The proper support of specification comprehension activities implies to provide developers and maintenance personnel with partial specifications that

- are substantially smaller than the full specification,
- contain all relevant parts of interest, and that

$[NAME, DATE]$

BB

$known : \mathbb{P}\, NAME$
$birthday : NAME \pfun DATE$

$known = \mathrm{dom}\, birthday$

Add

ΔBB
$name? : NAME$
$date? : DATE$

$name? \notin known$
$birthday' = birthday \cup \{name? \mapsto date?\}$

Figure 12.1 Fragment of the Z Birthday-Book specification out of [7]. It specifies a system for storing names ($NAME$) and birthday dates ($DATE$) in a database called BB. The Add operation schema takes a name and a date as arguments and stores them in the database.

- can be automatically derived from the original specification.

The informatics literature is full of concepts of partiality, concepts that aim to provide an interested party just the perspective needed for a particular task. The notions of views [8], slices [16, 17], and chunks [6] come to mind, and it seems obvious to map those concepts to specifications.

Specifications and programs are different and mapping decomposition concepts from programming languages to declarative languages is a challenging part. There is an explicit flow of control with imperative programs. Line-numbers represent an explicit order among statements or other constructs and it is state of the practice to apply techniques like constructing a PDG for further dependency analysis.

On the other hand, when looking at declarative specification languages like VDM [9] or Z [14], there is no explicit flow of control. The notion of control dependency has to be re-interpreted. Thus, when talking about the decomposition of a specification, one has to be careful in putting specifications on a par with imperative programs.

As the assessment of the approach is based on the notions of specification slices and chunks, these concepts are presented in the remainder of this section. Readers interested in a more formal set of definitions are referred to [3, 4].

12.2.1 Elements of Formal Specifications

Formal specifications are expressions written in some formal language. Each specification language consists of linguistic elements, and the primitives of the language have a formally defined semantics. These primitives are referred to as *literals* of the language. Examples of

literals are identifiers or linguistic operators, such as the literal "$\mathbb{P}$", the identifier "$NAME$" or the operator "dom" in the birthday book specification that is represented in Fig. 12.1.

Minimal and meaningful linguistic expressions are called *prime objects* or *primes* for short. They are syntactic elements of specifications with a formally defined semantics. Such primes are constructed from literals and form the basic entities of specifications. Examples are declarations like "$name? : NAME$" or expressions such as "$name? \notin known$" in the Add operation schema of the specification in Fig. 12.1.

One important aspect is that primes are immutable as far as they constitute fundamental units (e.g. states and operations) that specifications are built upon. Several specification languages permit semantically richer primes (the Add operation schema is such a fundamental unit). These primes are called *higher level primes* and are well-defined arrangements of literals and other primes.

It is not sufficient to support the comprehension process by just looking for independent prime objects. During comprehension tasks the user of a specification needs more than a given prime but less than the full specification. As the needed information is distributed on the two-dimensional, linear text, one needs a set of different primes which are to be found in different parts of the specification. Such a set, consisting of different but related primes is called *specification fragment*. Such a fragment might be the unboxed Z paragraph "$[NAME, DATE]$" and the set consisting of the two primes "$known : \mathbb{P}\, NAME$" and "$birthday : NAME \pfun DATE$". This fragment of the specification in Fig. 12.1 grasps the state space of the specification without telling anything about the relationship between $known$ and $birthday$.

With these (informal) definitions, specification languages can be structured into elements having defined semantics on their own (primes) and elements which obtain their semantics from their arrangement in a broader context [11].

12.2.2 Specification Slices and Chunks

The previous section focused on syntactic components of specifications. In the scope of comprehension activities some higher-level abstraction with *clearly defined* semantics are needed and it makes sense to start constructing these abstractions from well-known specification elements, namely from specification primes. Such higher-level (or semantic-based) specification concepts are, among others, the already mentioned concepts of slices and chunks.

Both, slices and chunks, are abstractions of the given specification defined "around" a specific point of interest. In analogy to the slicing criterion of program slicing, this point of interest is called *abstraction criterion*. It denotes a specific prime or a set of primes of the specification.

The basic mechanism for the identification of slices and chunks is, as with most program comprehension approaches, based on the identification of control (and data) dependencies. However, state-based specifications are in general different from imperative programs. Control is not predominant, there are no line numbers and the ordering of predicates is irrelevant, thus primes are not "lined up" by flow of control.

When looking closer at specifications it can be observed, though, that parts of a specification are in essence controlled by other parts of it: post-conditions are dependent on

pre-conditions. In languages where pre-conditions are explicitly highlighted, this is evident. In other languages, such as Z, one may resort to theorem proving techniques to identify pre-conditions.

The calculation of the relevant pre- and post-conditions is a time-consuming task, but the calculation can be skipped by taking before-state predicates as pre-conditions and after-state predicates as post-conditions directly. Before-state and after-state identifiers can be identified on a syntactical basis efficiently. For Z it is shown in [3] that this syntactic approximation is quite accurate. In Fig. 12.1 the identifier "$birthday'$" in the Add operation schema denotes an after state (it is a primed identifier), and thus the predicate "$birthday' = birthday \cup \{name? \mapsto date?\}$" is a post-condition prime. As there is no after-state identifier in the predicate "$name \notin birthday$", this prime is said to be a pre-condition prime[1].

Control dependencies in Z are located in a schema between pre- and post-condition primes, but control dependencies are also to be found in schemata which are combined via schema operations. Here, again some approximation is conducted. The semantic analysis is skipped, and pre- as well as post-condition expressions are reduced to sets of before- and after-state primes.

By following these simplifications, definitions for control, data, and syntactical dependencies can be provided:

Definition 39 *A specification prime p is* **syntactical dependent** *on a specification prime q, if q is needed to keep p syntactically correct.*

Definition 40 *A specification prime p is* **control dependent** *on a specification prime q, if q potentially decides whether p applies or not.*

Definition 41 *A specification prime p is* **data dependent** *on a specification prime q, if data potentially propagates from q to p through a series of state changes.*

In the specification of Fig. 12.1 the prime "$birthday' = \ldots$" of the Add operation schema is control dependent on the prime "$name? \notin known$" as this prime decides whether the state is changed or not. More complex examples are presented in [3].

With the above definition of dependencies within specifications, a mechanism for carving out slices and chunks from formally correct specifications can be provided.

Definition 42 *A* **specification chunk (SChunk())** *is (i) a prime including all primes contained within it or, (ii) a set of primes that exists within the same specification scope. For each pair of primes within this set of primes either one prime is dependent on the other or both primes are dependent on a third prime (within the set of primes).*

The really important thing is that a chunk always has to be comprehensible in isolation. Compared to a specification fragment, a chunk contains enough (surrounding) context to stay understandable. That means that at least enough semantic information has to be taken into consideration when generating specification chunks. To denote those types of dependencies that are to be considered when calculating the specification chunk, the operation name is augmented by a set of characters representing the included dependencies (S..Syntactic-, C..Control-, D..Data-dependencies). The operation "$SChunk_{[SD]}()$" represents a static chunk with respect to syntactic (S) and data (D) dependencies.

[1] Formally it can be shown that this predicate really is part of the post-condition of the Add operation schema.

Definition 43 *A* **full static specification slice (FSSlice())** *is a syntactically and semantically correct specification which is the result of adding those primes to an (initially empty) specification which are directly or indirectly contributing to the abstraction criterion.*

In contrast to the definitions provided in [7], the approach constructs specification fragments in a bottom up manner. This assures that every abstraction derived by the approach has a well defined semantics. Starting with a prime (or a set of primes) as abstraction criterion, the abstraction has this prime's (or these primes') semantics. By aggregating further primes properly, the remaining specification fragment always has a well defined semantics.

As with program slices, it is required that a specification slice is a syntactically correct specification and that it is semantically complete. This is obtained, if *all* kinds of dependencies are included in the specification criterion, which means that a full static specification slice can be calculated by calculating the static specification chunk "$SChunk_{[SCD]}()$".

12.2.3 Augmented Specification Relationship Net

Specification languages have built in clues to convey semantics. As with programming languages, syntax *and* layout of specification languages support the process of creation, but these characteristics are detrimental for detecting programming-language-like dependency types. However, hidden structural properties of written specifications can be detected in analogy to state-of-the-art techniques of other disciplines:

- In differential calculus it is sometimes easier to *transform* an equation into another space, solve the equation there and perform a backward transformation.
- When dealing with programming languages, a program is *transformed* to an abstract syntax tree which enables the construction of a program dependency graph, and which ultimately facilitates program dependency analysis.
- In [18] it is shown that a transformation of concurrent logic programs to a graph is useful for dependency detection. Zhao et. al. also show that such a graph forms a suitable basis for metrics calculation.

Following these ideas, it seems appropriate to transform the specification into a graph in order to analyze and identify dependencies. What is needed is a structure that, on the one hand, fully replaces the original specification, and, on the other hand, eases the identification of dependencies. Additionally the structure should be isomorphic. This guarantees that a transformation and backward transformation is possible in any case, and that it is up to the user which representation s/he chooses for the problem at hand.

A suitable form of representation is that of an augmented net. It can be used to cope with syntactic AND semantic information.

Structural information is captured in a net called *Specification Relationship Net* (SRN). Vertices in the SRN represent primes of the specification, and arcs represent relationships among them. Fig. 12.2 demonstrates how a Z schema is transformed to the SRN representation. Every prime is mapped to a prime vertex, and every schema is enclosed between two special vertices: a start and an end vertex[2]. References to other schemata are expressed by vertices connecting the referring prime to the referred SRN block.

[2]As can be seen in Fig. 12.2 the arcs in the net are classified. There are sequential-, AND- ($\wedge$) and OR- ($\vee$) control arcs used to express different logical and sequential relationships between primes in the specification. See [3] for more details.

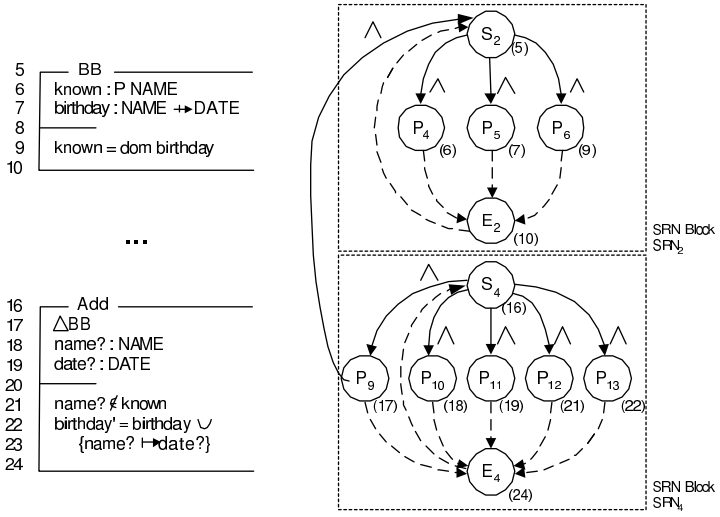


Figure 12.2 Z specification and SRN representing the birthday book state schema and the *Add* operation schema. Vertices are annotated by line numbers of the specification source. E.g. vertex P_4 represents the predicate prime "$known : \mathbf{P}\ NAME$" at line 6.

As specifications contain language- and layout-related information, the SRN is extended by vertices representing structural information and comments. This extension depends on the specification language at hand. The same holds for prime objects. The SRN is extended by this information and makes up the *extended SRN (eSRN)*. The transformation to the eSRN has two advantages:

- The eSRN can be used to deal with any information that exists in a specification. The SRN handles every prime and represents the "loose" structure of a specification. The eSRN is able to handle layout information and comments.

- The transformation function itself can be defined in a *bijective* manner. This means that a backward transformation is possible and the two forms of representation can be used interchangeable.

The last step is the augmentation of the net. To ease the identification of dependencies, the eSRN is augmented by declaration, definition and use information of identifiers attached to prime vertices. The ASRN captures the explicit semantics of the specification. Based on reachability conditions, the ASRN is then used to define control, data and syntactical/declarational dependencies in Z specifications. These dependencies then provide the basis for the definition of slices and chunks.

The net itself represents more than just a hidden structure of a specification. It reveals quite a lot of properties of the specification: its size, inter-relationships between primes and number and type of dependencies. It is a candidate for calculating specification metrics and provides the basis for the description of a specification's complexity.

12.3 The Complexity of Specifications

There are at least two possibilities to assess the effectiveness of the approach of generating and using specification fragments and thus raising their usability for comprehension activities. Firstly, by describing the effects on the complexity of the underlying specification (and therefore on the underlying task) via suitable metrics. Secondly, by conducting empirical studies.

These approaches are not mutually exclusive. However, even in the case of empirical studies, metrics are necessary for the assessment. This section suggests to use the ASRN in order to simplify the calculation of complexity measures and to facilitate the comparison between full specifications and their corresponding specification fragments. The calculation of ASRN related complexity measures is independent of the specification language. However, the approach is applied and validated on the basis of Z specifications.

It is generally agreed that a single measure is not sufficient to represent the overall complexity of a specification. The few existing size-based measures (like lines of specification code or numbers of operators) are not suitable to describe the complexity in its entirety as it is not only size that matters. This argument is easily reinforced as specifications of equal size are not necessarily of the same complexity. It is *logical and structural* complexity that is not to be neglected. The existing set of size-based measures can be extended by looking at the structural information that is explicitly available in the ASRN.

The following complexity measures are easily derived from the augmented specification relationship net:

- Conceptual Complexity $CC(\Psi)$ of a specification Ψ. Here, prime objects of the specification are counted instead of the number of lines of specification code.
- Extended Cyclomatic Complexity $v'(\Psi) = (v(l), v(u))$ of a specification Ψ. $v(u)$ is the upper bound of the tuple and represents the total number of control dependencies in the ASRN. $v(l)$ is the lower bounds and represents the number of pre-condition primes in the specification. This measure can be compared to the extended cyclomatic complexity of programs [10].
- The Definition/Use count metric $DU(\Psi)$ (in analogy to the DU Count metrics presented in [15]) of a specification Ψ. By counting the number of data dependencies the maximum number of data relationships in the ASRN are identified.

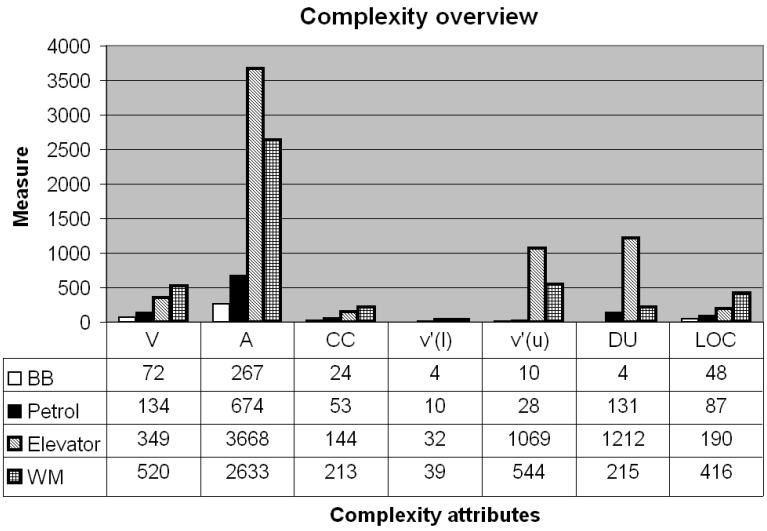
With these extensions to the ordinary set of measures detailed comparisons between specifications in respect to several aspects of complexity become feasible for the first time.

12.4 Specification Comprehension

The approach of calculating specification fragments promises to scale down the complexity of specifications and with it the complexity of the underlying comprehension task. This section provides several small experiments to underpin this statement.

12.4.1 General Setting

The birthday-book specification is one of the smallest specifications that are to be found in textbooks introducing Z. Larger specifications are necessary in order to express the effects



	V	A	CC	v'(l)	v'(u)	DU	LOC
BB	72	267	24	4	10	4	48
Petrol	134	674	53	10	28	131	87
Elevator	349	3668	144	32	1069	1212	190
WM	520	2633	213	39	544	215	416

Figure 12.3 Complexity overview regarding four Z-specifications. The table summarizes the total number of vertices V and arcs A in the ASRN representation, the conceptual complexity CC, the extended cyclomatic complexity $v' = (v'(l), v'(u))$, the DU (Def/Use) count metric, and the LOC of the specification.

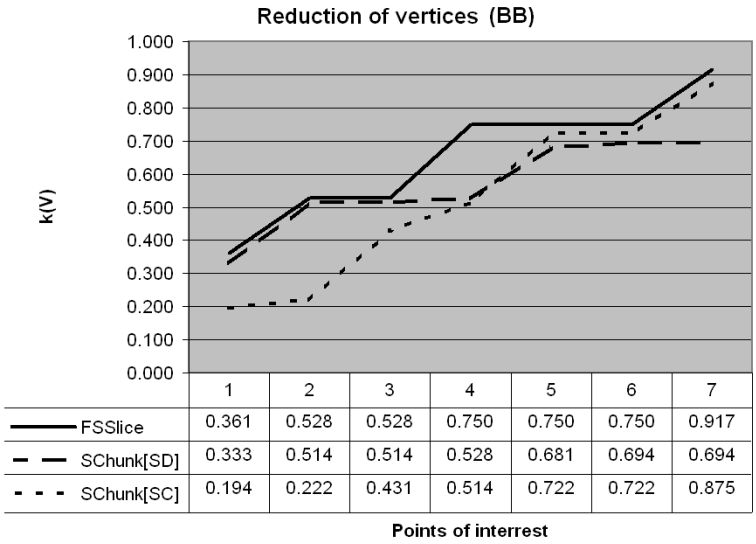
of specification fragments. This includes the influence of the different types of abstraction, the effect of the approach onto complexity in general and the effect of larger specifications.

This section makes use of three additional specifications: The "Elevator" specification out of [7], the "Petrol-Station" specification ($Petrol$ for short) which is used during class labs at the University of Klagenfurt and the "ITC Window Manager"-specification (WM for short) which is a commercial specification presented in [5].

Fig. 12.3 summarizes the key attributes of the specifications and visualizes the measures that contribute to their complexity: the total number of vertices (V) in the ASRN, the total number of arcs (A) in the ASRN, the extent of conceptual complexity (CC), the lower and upper bound ($v'(l), v'(u)$) of the extended cyclomatic complexity and the definition/use count (DU) of the specification.

As can be seen in Fig. 12.3, the BB-specification is the most simple one. The $Petrol$-specification is also small, but contains twice as many primes (and about twice as many vertices and arcs). The BB-specification consists of 72 vertices, the $Petrol$-specification consists of 134 vertices. The same ratio holds for the other measures, except for the DU count metric. Here, the $Petrol$-specification contains 131 data-dependencies, the BB-specification contains only 4 data-dependencies.

With respect to the total number of vertices in the ASRN, the $Elevator$-specification is the next larger one. The ASRN contains 349 vertices and is thus smaller than the WM-specification. It also contains only 144 primes, whereas the WM-specification consists of 213 primes. However, the $Elevator$-specification contains much more hidden dependencies. It contains 1069 control dependencies, whereas the WM-specification contains only 544 control dependencies. Thus, with respect to primes, the $Elevator$-specification is less complex than the WM-specification. With respect to the extended cyclomatic complexity



	1	2	3	4	5	6	7
FSSlice	0.361	0.528	0.528	0.750	0.750	0.750	0.917
SChunk[SD]	0.333	0.514	0.514	0.528	0.681	0.694	0.694
SChunk[SC]	0.194	0.222	0.431	0.514	0.722	0.722	0.875

Figure 12.4 Reduction factor k of vertices after the generation of all 7 specification fragments ($k = V_{fragment}/V_{orig}$) of the BB specification. Generating a slice in respect to prime 1 reduced the total number of vertices ($= 72$) to 26 – which results in a factor of $k(V)_1 = 0.361$.

and even the DU metric, the WM-specification is less complex. The same differences can be observed when looking at the number of lines of specification code.

12.4.2 Experiments

When tracing specifications it depends on the (reverse engineering) problem at hand which primes are relevant to the user and which types of fragments are to be generated. In order to provide a reliable statement about the benefits of the approach an experiment (and no case study) has been designed. The underlying idea is to look at a EVERY prime of the specification and therefor to look at all possible points of interest and all types of specification fragments.

A simple prototype has been implemented and used in order to generate all possible sets of specification fragments for every predicate prime of a given specification. The prototype takes a Z specification (type set in LaTeX) and a slicing/chunking criterion as input. It then generates the specification fragment and (i) exports the fragment into a LaTeX file and (ii) provides feedback about the ASRN representation.

The BB-, $Petrol$-, $Elevator$ and WM-specifications are used as experimental objects for the treatment – the application of the generation of specification fragments. For every point of interest three different types of specification fragments are generated.

In fact, the following steps are conducted during the treatment:

i. For *every* prime vertex representing a point of interest in the specification the full static specification slice $FSSlice()$ is calculated.

ii. For *every* prime vertex representing a point of interest in the specification the full static specification chunk $SChunk_{[S,D]}$ is calculated.

iii. For *every* prime vertex representing a point of interest p in the specification the full static specification chunk $SChunk_{[S,C]}$ is calculated.

For all points of interest slices and two types of chunks (one containing control dependencies, one containing data-dependencies) are calculated. In the BB-specification there are 7 points of interest. The $Petrol$-specification contains 21, the $Elevator$-specification contains 102, and the WM-specification contains 92 points of interest. Fig. 12.4 demonstrates the influence of the generation of specification fragments on the number of vertices in the ASRN representing the birthday book specification.

Depending on the specification, this leads to 21 (= 7 points of interest · 3 types of specification fragments) possible fragments for the BB-specification, 63 fragments for the $Petrol$-specification, 306 fragments for the $Elevator$-specification and 276 specification fragments for the WM-specification.

The factors which are assumed to change (and which are the response variables) are the following:

- Attributes regarding the ASRN. They include the total number of vertices V and arcs A of the net.
- The attribute contributing to the information content of the specification: $CC(\Psi)$, the conceptual complexity.
- Three attributes contributing to the logical complexity of the specification. This includes the lower bound l and the upper bound u of the extended cyclomatic complexity v' and the DU metric of the specification.

These attributes are determined for every specification fragment that is generated during the treatment. For the scope of the evaluation of the approach, the mean values of these attributes (in respect to a specific type of fragment) are considered.

12.4.3 Specification Fragments' Efficiency

12.4.3.1 Extent of Reduction of Complexity When generating all types of specification fragments, the first observation is that slices and chunks reduce the size of the specification. As slices include all types of dependencies and as chunks omit dependencies in the resulting specification, it seems to be obvious that, for a specific point of interest, a specification chunk should be smaller than the corresponding specification slice. Fig. 12.5 visualizes the results of the application of the slicing and chunking approach for all four specifications. It can be observed that there is a reduction of complexity in all cases – which is indicated by values of $k(...)$ that are lower than 1.

Additionally, it can be observed that the values of the reduction factors of chunks are in all cases lower than the values of the reduction factors of the corresponding slices. For the conceptual complexity CC of the BB specification the extent of reduction is $k(CC) = 0.542$ (when generating slices), whereas the extent of reduction is $k(CC) = 0.387$ (or $k(CC) = 0.435$) when generating chunks. The disadvantage of neglecting information becomes an advantage as the focus gets sharper.

12.4.3.2 Influence of the Specification's Size When comparing the extent of the reduction of complexity between the BB and the WM specification it can be observed that the reduction factors k of the WM specification are generally lower than the reduction factors k

Spec.(Deps.)	V	A	k(V)	k(A)	k(CC)	k(v'(I))	k(v'(u))	k(DU)	CC
BB (SCD)	72	267	0.655	0.555	0.542	0.679	0.486	0.571	**24**
BB (SD)	72	267	0.565	0.429	0.387	0.286	0.129	0.286	**24**
BB (SC)	72	267	0.526	0.438	0.435	0.571	0.229	0.071	**24**
m(k)			**0.582**	**0.474**	**0.454**	**0.512**	**0.281**	**0.310**	
M(k)			**0.528**		**0.389**				
Petrol (SCD)	134	674	0.709	0.661	0.686	0.829	0.816	0.685	**53**
Petrol (SD)	134	674	0.631	0.510	0.556	0.138	0.077	0.441	**53**
Petrol (SC)	134	674	0.242	0.118	0.147	0.200	0.071	0.004	**53**
m(k)			**0.527**	**0.430**	**0.463**	**0.389**	**0.321**	**0.377**	
M(k)			**0.478**		**0.388**				
Elevator (SCD)	349	3668	0.828	0.752	0.753	0.915	0.778	0.735	**144**
Elevator (SD)	349	3668	0.548	0.194	0.293	0.039	0.005	0.140	**144**
Elevator (SC)	349	3668	0.315	0.071	0.164	0.358	0.011	0.002	**144**
m(k)			**0.564**	**0.339**	**0.403**	**0.437**	**0.264**	**0.293**	
M(k)			**0.451**		**0.349**				
WM (SCD)	520	2633	0.454	0.364	0.359	0.529	0.307	0.376	**213**
WM (SD)	520	2633	0.176	0.060	0.061	0.031	0.003	0.047	**213**
WM (SC)	520	2633	0.185	0.070	0.088	0.168	0.012	0.002	**213**
m(k)			**0.272**	**0.165**	**0.169**	**0.242**	**0.107**	**0.141**	
M(k)			**0.218**		**0.165**				

Figure 12.5 Table summarizing the mean values m of the reduction factors with respect to four specifications and three different types of abstractions. Additionally it provides the mean M of the reduction of ASRN and specification attributes. The first column names the specification and the corresponding abstraction criteria. An SCD indicates the generation of a specification slice, SD and SC indicate the generation of a specification chunk.

of the BB specification (which means a higher extent of reduction). Calculating the values for specifications of different sizes leads to a reasonable observation: the effect obtainable by slicing and chunking rises with the size of the specification under consideration.

The assumption is based on the heuristic that slicing and chunking carve out concepts of the specification, concepts that are independent from the specification at hand and are of (roughly) equal size. Thus it is reasonable that with larger specifications the extent of the reduction increases, too.

When talking about the size of a specification, the conceptual complexity (which is equivalent to the number of primes in the specification) is an appropriate measure. However, to describe the average case, the mean values (M) of all reduction factors are taken as a basis and Fig. 12.5 summarizes the dependent and independent variables. It can be observed that the mean values M, expressing the average reduction of ASRN and specification attributes, decrease with increasing size of the specifications.

Fig. 12.6 summarizes the mean M of the mean values of the reduction factors for ASRN metrics and complexity measures. It can be observed that the larger the specification the higher the overall reduction. The ASRN of the BB-specification ($CC = 24$) is reduced by a factor of 1.89 ($= 1/0.528$). The logical complexity is reduced by a factor of 2.57 ($= 1/0.389$). This can be compared with the much larger WM-specification ($CC = 213$). Here, the ASRN net is reduced by a factor of 4.59 ($= 1/0.218$). The logical complexity is reduced by a factor of 6.06 ($= 1/0.165$). The mean value of the reduction increases with increasing size of CC, thus the approach is more efficient when applied to larger specifications.

Nevertheless, the basic assumption that both approaches carve out concepts from a specification in a similar way does not hold. Slices ensure that all dependencies are considered, chunks allow to neglect existing information. This means that when generating chunks

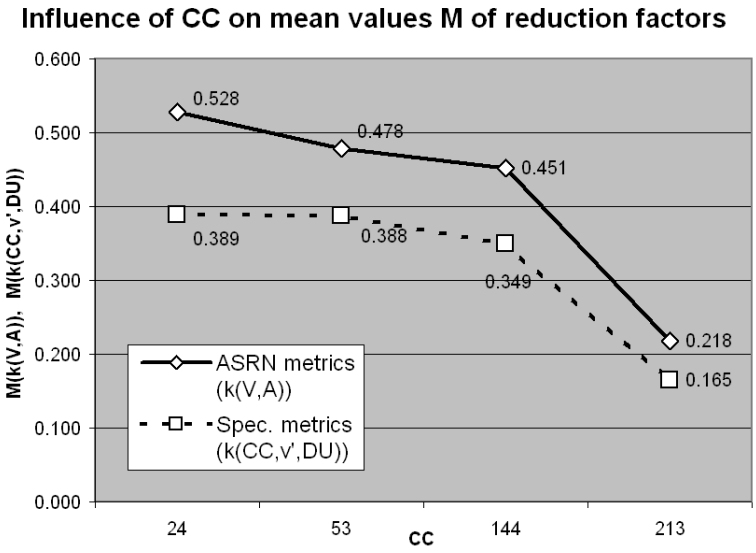


Figure 12.6 Influence of size on the extent of reduction. The values are based on the results summarized in Fig. 12.5 and demonstrate that with increasing size of the underlying specification the extent of reduction increases, too.

only, the chance is higher to generate smaller specifications. With slices one might have similar problems as with program slicing – the resulting slice has to contain all necessary statements and it is thus as large as the original program.

Generally speaking, the larger the specification, the higher the mean value of the reduction factor of the net and the mean value of the reduction of a specification's complexity. When considering only one type of abstraction, this observation does not hold. Taking a closer look at the generated specification slices, the value of $k(V)$ *increases* with increasing size of the specifications.

12.4.3.3 Efficiency of the Approach Two aspects are important when generating slices or chunks: the benefits should increase with increasing size of the specifications at hand. This means that the generated specification fragments should stay at a comparable conceptual level. Then, logical complexity should decrease significantly. The effect of scaling down specifications should influence the reduction of hidden dependencies at least with equal size.

For the first aspect not the extents of reduction, but measures contributing to the information content are of interest: the total numbers of vertices (V) and arcs (A) in the ASRN and the conceptual complexity CC.

For the second aspect the relationship between conceptual and logical complexity is of interest. Vertices represent conceptual entities and arcs represent logical dependencies. Thus the comparison of conceptual and logical complexity can be carried out by looking at the ratio between the reduction of vertices ($k(V)$) and the reduction of arcs ($k(A)$) in the net. This ratio is described by the factor $f(k)$ which is defined as follows: $f(k) = k(A)/k(V)$. It expresses the ratio between the decrease of vertices and the decrease of

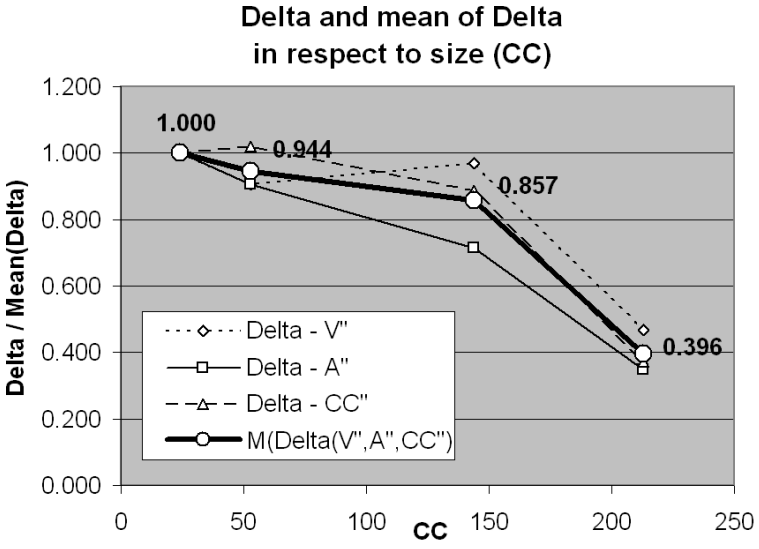


Figure 12.7 Effect of reduction when increasing the size (CC) of the specification. If the effect increases at the same ratio as the original specification, then the value of *Delta* should be 1. A value of *Delta* lower than 1 indicates that the approach is more efficient.

arcs in the net. A value lower than 1 indicates that arcs are decreased to a greater extent than vertices. This leads to the following definition:

Definition 44 Efficiency of the approach. *The approach of generating specification fragments is treated* efficient*, iff*
(a.) the mean values of the size of the net and the mean values of the complexity increase at a lower scale than the original specifications.
(b.) the logical complexity (described via v' and DU) is at least reduced to the same extent as the conceptual complexity (described via CC).

The basic assumption is that the reduction of the number of vertices leads to a higher reduction of arcs in the net. Formally, when the ASRN contains n vertices and a arcs, then there are $3 \cdot n^2$ possible arcs (of types C, D and S), at the most, in the net. If decreasing the number of vertices n by a factor of k_v, then the number of arcs should decrease at least by a factor k_a that is equal or higher than k_v. It holds:

$$
\begin{aligned}
& a = O(3 \cdot n^2) \\
& a' = O(3 \cdot (n/k_v)^2) \qquad\qquad with\ k_v = O(n/(n')) \\
& k_a = a/(a') = O(\tfrac{3 \cdot n^2}{3 \cdot (n/k_v)^2}) = O(k_v^2)
\end{aligned}
$$

In the optimal case the reduction factor k_a increases with the square of the reduction of the factor k_v. This simple heuristic is the basis for the assumption that the decrease of the number of dependency-arcs in the net (which influence logical complexity) is at least as high as the decrease of the number of vertices.

The following observation can be made: The mean value of the sizes of specification fragments (described by a factor $Delta$ which is dependent on V', A', CC') increases at

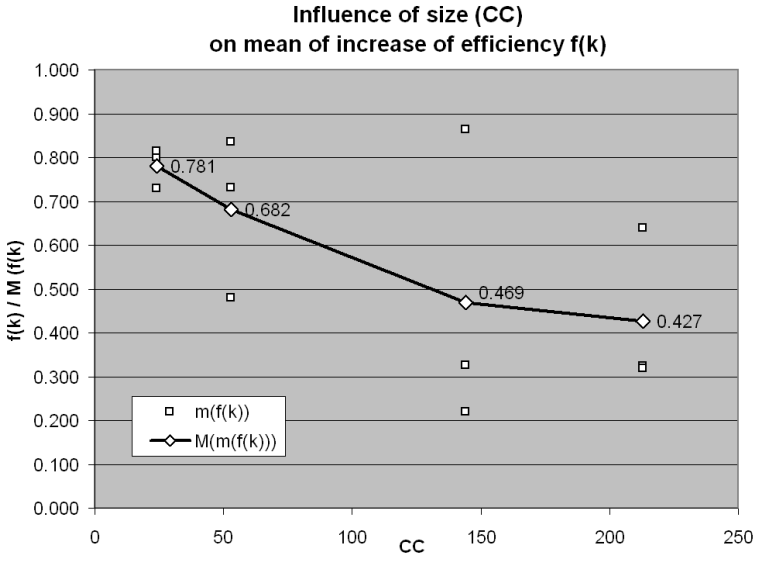


Figure 12.8 The effect of reduction ($m(f(k))$) increases with the size (CC) of the specification. If the effect increases at the same ratio as the original specification, then the value of $m(f(k))$ should be 1. Additionally, the mean of all reductions ($M(m(fk))$) is presented.

a lower scale than the sizes of the underlying specifications. An increase of CC by a factor I results in an increase of size and complexity by a factor less than I. It holds that $M(Delta) \leq 1$. Here, I describes the increase of the specification in respect to vertices, arcs and conceptual complexity. $Delta$ expresses the ratio between true values of reduction attributes and estimated values of that attributes. A value lower than 1 indicates that the extent of decrease is higher than the expected extent of decrease.

A value of $Delta$ less or equal than 1 indicates that the increase of complexity measures is less or equal than the increase of the size of the specification. This indicates that the approach is efficient in terms of Def. 44. The complexity of the specification fragments increases at lower scale than the size of the underlying specifications. If the value is about 1 then there is still reduction of complexity. However, this implies that, when the size of the specification increases by a factor of 2, the approach generates specification fragments that also increase by a factor of 2.

As can be seen in Fig. 12.7 there is a decrease in the value of $M(Delta)$ with growing sizes of the specification. Starting with the BB-specification the factor is 1.000. The $Petrol$-specification is about twice as large as the BB-specification (for the number of vertices $I = 1.8$). However, the fragment contains $V' = 70.667$ vertices in the mean, and not $\tilde{V}' = 77.989$ vertices. The approach has been a bit more effective. Therefore $Delta = 0.906$ which is less than 1. When looking at vertices, arcs and the conceptual complexity, the mean value $M(Delta)$ is 0.944 which is also less than 1. The same holds for the mean values $M(Delta)$ of the $Elevator$- and WM-specification.

The second part of the definition of efficiency deals with the ratio between logical and conceptual complexity. It can be observed that the logical complexity is at least reduced to

the same extent as the conceptual complexity. Thus when generating all possible fragments it holds that $M(f(k)) \leq 1$.

The factor $f(k)(= k(V)/k(A))$ tells a lot about the ratio between the reduction of vertices and arcs in the net. If $f(k)$ decreases, this indicates that the number of arcs is reduced to a much greater extent than the number of vertices. This underpins the heuristic that, in the average case, the reduction $r_A = 1/k(A)$ increases with at least the order of the reduction of the factor $r_V = 1/k(V)$.

When looking at the factor $f(k)$ for all four specifications it can be observed that all the values are less than 1. This indicates that, in any case, the extent of reduction of arcs is higher than the extent of the reduction of vertices. When looking at the mean of the reductions ($M(f(k))$) it can be observed that, with increasing size of the underlying specification, the extent of the reduction increases, too. The approach gets more and more efficient with larger specifications at hand. Fig. 12.8 visualizes the values for the factor $f(k)$ (dependent on CC) and the mean $M(f(k))$ of the factors. It also underpins the statement of the positive effects of the approach.

12.4.3.4 Summary of Observations Based on the four experimental objects all possible types of specification fragments for all points of interest have been calculated. The following observations have been made:

O1 Specification chunks reduce complexity more than specification slices. This observation holds for all specifications that have been examined so far. Additionally, there is strong evidence that chunks reduce the complexity in more cases than specification slices do.

O2 The effect obtainable by slicing and chunking rises with the size of the specification under consideration. For the mean values of complexity attributes this observation has been confirmed. However, there are a few cases when this observation does not hold. Again, slices do not always lead to smaller specifications, while chunks usually do.

O3 The effect observable by slicing and chunking is significant. It can be stated that the slicing/chunking approach decreases complexity to a much greater extent when specifications are getting larger. In fact, it can be shown that the mean value of the increase of complexity of the generated fragment is definitely less than the increase of the size of the specification.

12.5 Conclusion

Based on the argument that formal specifications are useful not only during initial software development but also during maintenance, this work presents specification fragments to help maintainers obtaining a focussed understanding of specifications.

More than 600 specifications fragments have been examined in order to demonstrate the efficiency of the approach. The experiments demonstrates that the approach can be used to achieve the goal of scaling down specifications in order to make them more comprehensible. It is shown that the generation of specifications fragments is sufficiently efficient.

The specifications used in this work represent typical specifications to be found in the Z-literature. However, much larger specifications (with several thousands of primes) are

still waiting to be examined. It is likely that the approach still proves useful. The results generated by the prototype imply that, with larger specifications, there is generally a trend toward higher extents of reduction of complexity.

REFERENCES

[1] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software Systems*. John Wiley & Sons, Inc., 1995.

[2] Andreas Bollin. Specification transformation as a basis for specification comprehension. In *Proceedings of Applied Informatics 02*. AACE, 2002.

[3] Andreas Bollin. *Specification Comprehension – Reducing the Complexity of Specifications*. PhD thesis, Universität Klagenfurt, April 2004.

[4] Andreas Bollin and Roland R. Mittermeir. Specification fragments with defined semantics to support sw-evolution. In *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'03)*. IEEE ArAb Computer Society, 2003.

[5] Jonathan Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press (ITCP), 1996.

[6] Ilene Burnstein, Katherine Roberson, Floyd Saner, Abdul Mirza, and Abdallah Tubaishat. A role for chunking and fuzzy reasoning in a program comprehension and debugging tool. In *TAI-97, 9^{th} International Conference on Tools with Artificial Intelligence*. IEEE press, November 1997.

[7] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[8] Daniel Jackson. Structuring Z Specifications with Views. *ACM Trans. on Software Engineering and Methodology*, 4(4), October 1995.

[9] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, 2^{nd} edition, 1990.

[10] Thomas J. McCabe. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1989.

[11] Roland T. Mittermeir and Andreas Bollin. Demand-driven specification partitioning. In *Proceedings of the 5th Joint Modular Languages Conference, JMLC'03*, Ausgust 2003.

[12] Tomohiro Oda and Keijiri Araki. Specification slicing in a formal methods software development. In 17^{th} *Annual International Computer Software and Applications Conference*, IEEE Computer Socienty Press, pages 313–319, November 1993.

[13] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *10th International Workshop on Program Comprehension (IWPC'02)*, June 2002.

[14] J.M Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 2^{nd} edition, 1992.

[15] Kuo-Chung Tai. A program complexity metric based on data flow information in control graphs. *Proceedings of the 7th International Conference on Software Engineering*, pages 239–248, 1984.

[16] Frank Tip. A Survey of Program Slicing Techniques. Technical report, CWI Netherlands, 1994.

[17] Mark Weiser. Program slicing. In *Proceedings of the 5^{th} International Conference on Software Engineering*, pages 439–449. IEEE, 1982.

[18] Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima. Program dependence analysis of concurrent logic programs and its applications. In *Proceedings of 1996 International Conference on Parallel and Distributed Systems*, pages 282–291. IEEE Computer Society Press, June 1996.

APPENDIX A

CURRICULUM VITAE

CURRICULUM VITAE

Andreas Bollin was born on October 16, 1971 in Graz, Austria. He attended secondary school (Realgymnasium Pestalozzi) from 1981 to 1989 where he passed the school leaving examination with distinction. He joined the Technical University of Graz, Austria, for his study in Telematics in October 1989. In 1993, he joined the Institute of Software-Technology (IST) at Graz University of Technology, where he became a member of the research team working on functional programming languages. At the same time he also worked as a technician at the IST. Between 1994 and 1998 he additionally worked for an Irish company (International Software Consulting Network Ltd.) as communication expert and leader of the software development team. In 1998, he got the position of a Graduate Assistant at the Institute for Information Systems and Computer Media (IICM), Graz University of Technology, where he started to teach programming languages (SML, CML, Java) and the specification language VDM.

In December 1999, he obtained his "Diplom Ingenieur" degree (Master of Science) in the field of Telematics under the tutorship of Prof. Peter Lucas. The thesis (in English) was titled "Dialogue classification and specification".

In April 2000, he moved to the University of Klagenfurt, Austria. He joined the working group of Prof. Roland Mittermeir at the Department of Informatics Systems (ISYS) where he also started with his doctoral studies in the fields of specification slicing. In 2001, he became a member of the research team of Planet-ET (Platform and Network for Educational Technologies) with a focus on E-Learning and curriculum development. Also in 2001, he became deputy leader of the AMEISE (A Media Education Initiative for Software Engineering) project that focussed on teaching software project management by simulation.

In April 2004, he received his Ph.D. with distinction in the field of Applied Computer Science from the University of Klagenfurt. The title of the Ph.D. thesis (in English) was "Specification Comprehension – Reducing the Complexity of Formal Specifications".

Andreas Bollin is currently Assistant Professor at the Software Engineering and Soft Computing group of the University of Klagenfurt, Austria, where he is still active in the fields of Software Comprehension and Reverse Engineering, Formal Methods, but also Didactics in Informatics and Multimedia Systems.