Andreas Bollin

Specification Comprehension
Reducing the Complexity of Specifications

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

Universität Klagenfurt
Fakultät für Wirtschaftswissenschaften und Informatik

1. Begutachter:	Univ.Prof. Dipl.-Ing. Mag. Dr. Roland T. Mittermeir
Institut :	Institut für Informatik-Systeme
2. Begutachter:	Univ.Prof. Dipl.-Ing. Dr. Martin Hitz
Institut :	Institut für Informatik-Systeme

April/2004

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Schrift verfasst und die mit ihr unmittelbar verbundenen Arbeiten selbst durchgeführt zu haben. Die in der Schrift verwendete Literatur sowie das Ausmaß der mir im gesamten Arbeitsvorgang gewährten Unterstützung sind ausnahmslos angegeben. Die Schrift ist noch keiner anderen Prüfungsbehörde vorgelegt worden.

(Unterschrift)

(Ort, Datum)

Dedication

Many thanks to my wife, who supported me on the long way of research with all her love. Without her and her knowledge about education (one strong motivation was the improvement of courses teaching formal methods) this work would not be as complete as it is.
The next person I want to express my deepest thanks to is my mentor, Roland T. Mittermeir.
This would indeed be an incomplete dedication if I did not thank my whole family, too. I did not always have as much time as I wanted to spend with them, and without their patience and understanding this work would never have been completed.

This work is dedicated to my beloved father who departed life too early.

ABSTRACT

Formal methods and the application of formal specification languages play a crucial role in software engineering. Several studies indicate their benefits, and the use of formal specifications raises the overall quality of the development process. Especially when specifications are kept up to date (during evolutionary steps), they play a vital role in the maintenance phase. Whether it is a myth or a fact, due to their semantical compactness specifications are often criticized to be hard to understand. To diminish this problem, it seems to be useful to apply comprehension approaches to specifications, to weaken their semantical compactness, to visualize implicit information and to support focusing on a specific point of interest.

Formal specifications remain compact structures, but in respect to maintenance and comprehension tasks the complexity of specifications can be reduced effectively. The objective of this work is to sustain the process of specification comprehension. This is achieved by reducing the complexity of specifications by focusing on those parts which are necessary to solve a specific problem at hand. Several factors contribute to the overall complexity. The vast majority of comprehension 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.

Typically as it is, the comprehension process is sustained by making implicit information explicit. Therefore, this work suggests to visualize dependencies that are hidden in the specification. It introduces a specification representation in the form of an augmented specification relationship net (ASRN for short).

The identification of dependencies is important not only for visualization aspects. Similar to program comprehension approaches the calculation of specification slices and chunks also depends on the identification of control-, data- and syntactic dependencies. However, programming-language-like notions of dependencies are not applicable for specification languages. Thus this work introduces the notions of control-, data- and syntactic dependencies in specifications. These dependencies are efficiently calculated by using the ASRN and they form the basis for the generation of specification slices and chunks.

Together with the visualization of dependencies several comprehension tasks are supported. Partial specifications help to focus on a specific point of interest, and

feedback concerning the structure of the specification speeds up the overall comprehension process. Generally speaking, the approach aims at reducing the perceived complexity of the specification during comprehension and maintenance tasks.

Complexity is measured by a wide range of attributes. Well-known complexity measures are adopted to specifications, and based on these measures the approach is evaluated in respect to its usefulness. It can be shown that the generation of partial specifications is very efficient and effective. This work demonstrates that, with an increasing size of the specification at hand, the reduction of complexity (achieved by the generation of partial specifications) increases to a greater extent, too.

The evaluation has been realized by a small prototype which is able to generate partial specifications out of Z. The studies demonstrate that the approach can be used to achieve the goal of making specifications more comprehensible. With all these findings at hand formal specifications are no longer hard to understand.

ZUSAMMENFASSUNG

Formale Methoden spielen eine wichtige Rolle in der Welt des Software Engineerings. Anfänglich eingesetzt um die Qualität des Endproduktes zu steigern, bzw. um Testdaten teils automatisch zu generieren, zeigt sich bald, dass formale Spezifikationen eine wertvolle Rolle während Wartungsprozessen spielen. Dieser Vorteil wird teilweise wettgemacht durch den Ruf, der formalen Spezifikationen vorauseilt: Spezifikationen sind zu komplex und daher schwer zu verstehen.

Diese Arbeit beteiligt sich nicht an der scheinbar endlosen Diskussion über Vor- und Nachteile von Spezifikationen, sondern präsentiert einen Ansatz um die wahrgenommene Komplexität einer Spezifikation zu reduzieren. Durch das Unterstützen des Verständnisprozesses und durch das Verringern der Komplexität sollte den vielen Mythen um die Nachteile formaler Methoden entgegengewirkt werden.

Spezifikationen sind komplexe Gebilde und mit wachsendem Umfang der Spezifikationen sind diese tatsächlich schwerer zu verstehen. Neben dem Umfang sind es vor allem auch versteckte Abhängigkeiten im Spezifikationstext, welche den Verständnisprozess erschweren. Es ist also angebracht, einerseits den Umfang zu reduzieren, und andererseits diese Abhängigkeiten explizit darzustellen.

Im Fachgebiet des Programm-Verstehens gibt es bekannte Konzepte um den Umfang zu reduzieren, aber auch um sicherzustellen, dass relevante Informationen nicht verloren gehen: es sind dies unter anderem die Konzepte der Erzeugung von Slices und Chunks. Diese Arbeit greift diese Konzepte der Partialität auf und definiert Slices und Chunks für Spezifikationen.

Auf der anderen Seite gibt es in Spezifikationen versteckte Abhängigkeiten, und hier wird, analog zu herkömmlichen Ansätzen des Programm-Verstehens, vorgeschlagen, die Struktur und explizit bzw. implizit vorhandene Abhängigkeiten in einem annotierten Netz darzustellen. Dieses Net wird ASRN (Englisch für "Augmented Specification Relationship Net") genannt.

Die Identifizierung dieser Abhängigkeiten, es handelt sich hierbei um Kontroll-, Daten- und Syntaxabhängigkeiten, ist nicht nur für die Darstellung der Struktur von Bedeutung. Diese Abhängigkeiten sind auch für die Erzeugung von Slices und Chunks essentiell. Erschwert wird die Analyse der Abhängigkeiten jedoch durch die deklarative Natur von vielen Spezifikationssprachen. Kontroll-, und Datenabhängigkeiten sind nicht so einfach wie in der herkömmlichen, imperativen Welt

der Programmiersprachen zu identifizieren. Diese Arbeit definiert daher Kontroll- und Datenabhängigkeiten in einer für die Analyse brauchbaren Weise. Weiters wird eine Methode vorgestellt, um diese Abhängigkeiten effizient anhand des ASRNs zu identifizieren.

Im Rahmen dieser Arbeit wurde ein einfacher Prototyp implementiert welcher in der Lage ist, aus Z-Spezifikationen partielle Spezifikationen zu erzeugen. Der vorgestellte Ansatz ist in der Lage die Komplexität einer Spezifikation unter dem Blickpunkt einer bestimmten Aufgabe deutlich zu reduzieren. Die im Rahmen dieser Arbeit durchgeführte Evaluation des Ansatzes zeigt, dass mit zunehmender Größe der Ausgangs-Spezifikation das Konzept der partiellen Spezifikation sehr effektiv ist. Gängige Komplexitätsmaße werden hierzu auf Spezifikationen angewandt und herangezogen, um diese Aussage in Bezug auf die Verwendbarkeit des Konzeptes zu untermauern.

Der Umfang von Spezifikationen kann für bestimmte Aufgaben deutlich (messbar) verringert werden. Der Verständnisprozess wird erleichtert und dem Mythos der unverständlichen Spezifikation somit entgegengewirkt.

CONTENTS

SELECTED LIST OF DEFINITIONS

SELECTED LIST OF RULES

1. THOUGHTS ON SPECIFICATION COMPREHENSION

Natura semina nobis scientiae
dedit, scientiam non dedit.

Seneca op. 120,4

1.1 The World of Specifications

Probably all of us remember the sentence: "Computers do not make mistakes". But before we allow ourselves to be lulled into a false sense of security, we also have to take into consideration that computer software is written (like hardware systems are designed) by humans – humans, who certainly do make mistakes[1].

The fact that human beings are prone to make mistakes is not the only problem. Software is also growing in size and complexity at a good pace. Thus, it is no wonder that software developers started to look for techniques to describe software requirements as clearly as possible, trying to reduce natural languages' ambiguity.

One attempt to do so resulted in the use of so called *Formal Methods*, a term which initially comes from formal logic. It now describes the notion of using languages and tools with a formal, mathematical basis in order to formally write down (and prove properties of) a system's specification. A specification is information about properties required by a (piece of) software. According to IEEE ([IEE91], as cited in [Tuc96, p.2303]), the term *Formal Specification* is defined by the following two alternative definitions: a formal specification is

1. a specification written and approved in accordance with established standards.

2. a specification written in a formal notation, often for use in proof of correctness.

In fact, the latter definition stresses one of the major advantages of formal specifications: the formal *notation*. In most cases this notation is based on mathematical algebra. The notation has a *well defined semantics* which allows the expression of specifications in an unambiguous way and supports abstracting away from implementation-related details. Jonathan Bowen and Michael Hinchey [BH97] name

[1] For more information see the Human Errors Website, http://Panko.cba.Hawaii.edu/HumanErr maintained by Ray Panko. Last visited: Nov. 2003.

four very good reasons for the use of mathematics as a basis for a formal notation: precision, conciseness, abstraction, and reasoning.

Whereas precision and conciseness can be naturally expected when using mathematics, it is abstraction which allows concentrating on the essential features of the system. Mathematics enables reasoning about systems' properties, but it is again abstraction which is the key idea to make formal methods a success. For handling different levels of abstractions, several specification languages and various "dialects" have evolved. According to [AP98] and [Mye97, p.2305], formal specification languages can be divided into four major classes:

- Property-oriented approaches. Algebraic specification languages, first-order logic and functional languages belong to this class. They can also be divided into approaches which use axiomatic semantics (e.g. Larch [GH83] or Anna [LvH85]) or algebraic semantics (e.g. OBJ [GWM+93]). The first makes use of first-order predicate logic to express pre- and post conditions of operations which are defined over abstract data types. The second one is based on multisorted algebras and describes properties of the system via mathematical equations.

- Model-oriented approaches. These approaches are based on a specific model of the desired system's behavior. Both, model and behavior are described by abstract mathematical objects. Hoare-logic and first-order logic (in combination with pre- and post-conditions) are used to express features of the system. Z [Spi89b] or VDM [Jon90] are examples of languages following the model-oriented approach. VDM provides the explicit notion of pre- and post-conditions, the syntax of Z on the other side does not differ between pre- and post-conditions. (However, they can at least be calculated using special operations.)

- Process algebras. They expound the behavior of a system by describing the algebras of their communicating processes. CSP [Hoa85] and CCS [Mil89] are examples of languages belonging to this class.

- State-machine-oriented approaches. Petri nets [Pet62, Rei85] and state charts [Har87] are examples of notations belonging to this class. However, the exact semantics and the reasoning aspects are not as well defined – facilitating, in some cases, further misunderstanding.

Abstraction, in combination with structuring facilities, provide mechanisms that allow a writer to surmount even large specifications. But it also turns out that even well-structured specifications are difficult to comprehend. Brooks (already in his 1975s edition of the Mythical Man Month) sums up this weak point [FPB95, p.63] with his typical ease:

> The formal definitions [...] have inspired wonder at their elegance and confidence in their precision. But they have demanded prose explanations to make their content easy to learn and teach.

Are formal specifications still too complex? It the content not easy to understand? Do they need further reinforcement in order to be comprehendible?

The point is that abstraction techniques are used to acquire a grasp of the complexity of software systems. Even abstract descriptions tend to get complex. And with increasing size they get more difficult to understand.

1.2 Comprehending Complex Systems

The use of formal methods is not uncontested (as will be explained in Chap. 2), but it is at least recommended as a means to produce high-quality software or it is even compulsory for the construction of high-level security or life-critical systems. It is beyond controversy that specifications can be (and are) used for test data generation and provide the basis for the verification of (at least parts of) their implementation.

As is elaborated in more detail in [BM03], formal methods only make sense if specifications are kept up to date during various evolutionary steps already taking place during system development, less to say later, during operation and maintenance. Keeping formal specifications up to date requires effort, and this effort is only justified if additional benefits can be gained from it, be it by speeding up software comprehension or by assuring higher quality software.

One of the key ideas of formal specifications is abstracting away from details of the implementation and writing down the nitty-gritty of the system in a very compact way. However, this density of expressing thoughts becomes detrimental for comprehending specifications in later phases [MB03]. Specifications are also getting large. The specification of the air traffic control system CDIS (Central Control Function Display Information System [Hal96]) which was developed by Praxis in the early 1990s, consists of about 1000 pages. Even abstract descriptions get complex if the size exceeds some limit. It is no wonder that managers claim that formal methods do not scale up and software developers look down on specification languages by referring to them as "write-only" languages. The solution is to reduce the size of the abstract description and thus to reduce the overall comprehension complexity.

We started with a well-known sentence which gave us some sense of (false) security, discovered that formal techniques have to be used in order to handle the complexity of software systems, and came to the conclusion that even formal specifications can get so complex that their deployment is impeded.

As will be discusses in more detail in Chap. 2, it is not possible to change the inherent density of specification languages. Though, one has to consider that comprehension problems are compounded if problem-inherent complexity is combined with complexity of size – latter can be reduced. To address this problem of complexity and to support the deployment of formal methods, this work elaborates on the following key question:

Is it possible to reduce the complexity of specifications?

In fact, as we will see later, this question is a bit too general. It is not satisfying to answer it per se with yes or no. At least three further questions will have to be dealt with in order to provide a more useful answer:

Q1 *What kind of abridgements are practicable for what problem?* This is an important question, as, by reducing complexity (and size), one is likely to change (parts or all of) the original formal specification. The crux of the matter is: by changing the specification, the semantics of the system is about to change, too – whereas the results still have to stay useful. There are different problems which have to be dealt with when fulfilling some reverse-engineering or maintenance tasks. Abridgements not only have to be meaningful – they also have to fit the problem.

Q2 *How can abridgements be achieved?* Quite often specification languages have rather compact notations and rich semantics. Even the operation on small parts of a specification can be a rather challenging task. In order to keep the remainder understandable, it is important to use well-defined operations for dissecting specifications.

Q3 *Can a reduction of complexity be expounded?* When talking about reducing complexity we are, in fact, still talking about abridging (disassembling) specifications. It is important to know about the opportunities different reduction techniques are providing. The only way to compare them (and the only way to notice effects) is the definition of suitable metrics.

As explained in Chap. 3, suitable mechanisms can be identified to reduce (disassemble) specifications. Thus, it is possible to limit the part of a specification a user (the author and/or a maintainer) needs to understand when trying to resolve some specific questions emerging during system development and/or maintenance. This work presents two approaches satisfying exactly these requirements. The objective of this work is not only to provide the basis for higher professionalism with respect to high quality specifications, but also to better assessment of the effort involved with an incoming change request, as well as higher quality of the maintenance process itself.

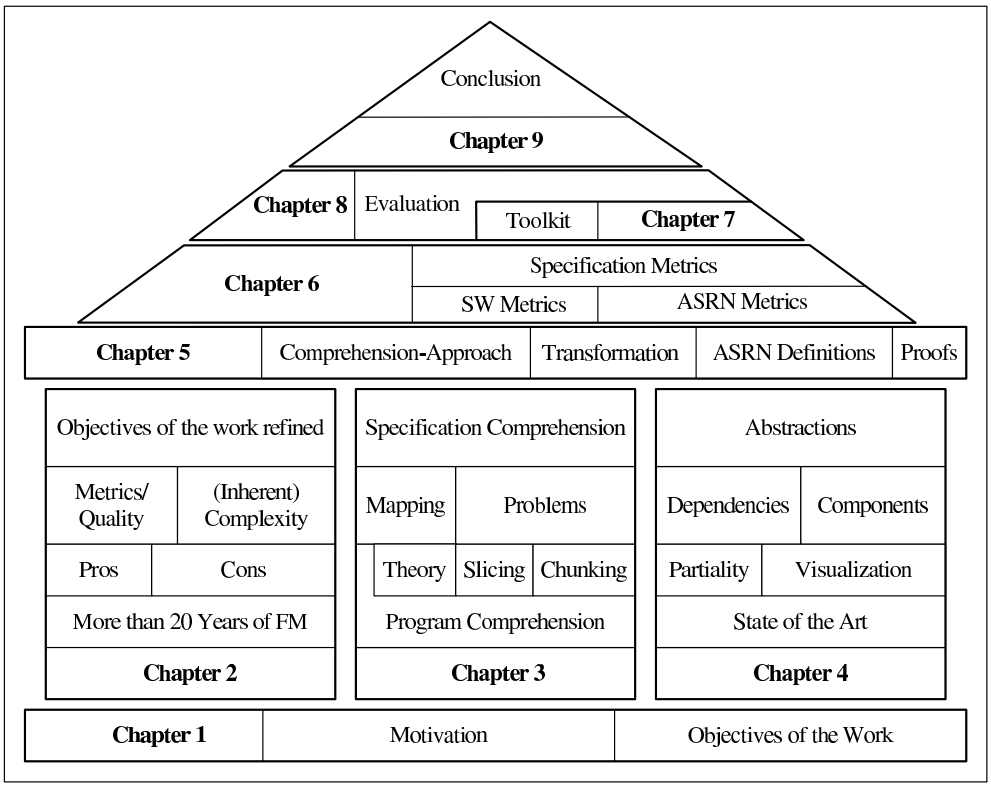


Fig. 1.1: Mapping of the basic structure of the work. Chap. 2 and Chap. 3 provide the theoretical background, Chap. 4 defines specification abstractions. Chap. 5 introduces an aid for visualizing and analyzing specifications. With that background, several complexity issues are discussed in Chap. 6. Chap. 7 introduces a small prototype and Chap. 8 presents several case studies. Finally an outlook and a conclusion are provided.

1.3 Organization of this Work

The structure of this thesis (see also Fig. 1.1) is as follows:

- Chap. 2 presents formal specifications in more detail and considers popular facts and fallacies. With this and the introduction of different complexity models, the basis for the understanding of specifications' complexity will be developed. The chapter concludes with a refinement of the questions raised in Chap. 1.

- Chap. 3 introduces the field of specification comprehension. This is done by first introducing and then comparing it to the field of program comprehension. The notion of dependencies, slicing and chunking is introduced and an

approach for the transformation of program comprehension techniques to the world of specifications is presented.

- Chap. 4 deals with different types of specification abstractions. Common abstraction techniques are presented and, depending on the desired or necessary context, these techniques are mapped to the world of specifications.
- Chap. 5 provides definitions of the so-called Augmented Specification Relationship Net, a graphical representation which serves as the basis for the analysis of specifications. As an example transformation rules for Z are given.
- Chap. 6 goes into the complexity of specifications, again. It develops different notions of specification metrics in order to provide appropriate tools for evaluating the above approaches.
- Chap. 7 describes a small Java-based prototype for slicing and chunking Z-specifications. This prototype provides the basis for the generation of slices and chunks of larger specifications and is thus fundamental for the evaluation of the approach.
- Chap. 8 evaluates the approach and introduces several case studies. It discusses benefits as well as drawbacks and limitations.
- Finally, Chap. 9 concludes the work and summarizes the findings.

The appendix provides background information: sample specifications are presented; a related bibliography section supplies links to further readings and a glossary summarizes and explains relevant terms.

2. COMPLEXITY OF SPECIFICATIONS

All difficulties
are but easy
when they are known.

W. Shakespeare [1564–1616]

There are many books and articles (see App. B) dealing with the question whether or under which circumstances the benefits of formal methods exceed the costs of converting to it. In fact a lot of aspects contribute directly or indirectly to the support or rejection of formal methods. The first section focuses on advantages, fallacies and impacts of applying formal methods during the software life-cycle. As it turns out, the inherent complexity of formal specifications and missing metrics hinder a broader deployment – so these two aspects are examined in more detail in the remainder of this chapter.

2.1 More than 20 Years of Formal Methods

The main objective of software engineering is to produce software that successfully works in the environment where it is intended to be used [Jac96, p.6]. What sounds rather trivial necessitates a rigorously planned software development process. Be it by following the rather systematic (Waterfall) model [Dav97] or be it by using some lightweight methodologies [WC03], developers have to ensure a uniform interpretation of requirements; they must be able to communicate effectively and precisely. Thus, the (requirement) specification has to be documented in a way that is unequivocally understood. Here, natural languages are expressive and necessary. But they are also imprecise. Informal descriptions (e.g. charts or diagrams) often do not even have an inherent and commonly understood semantics[1]. On the other hand formal approaches contribute a great deal to the quality of specifications and design documents. Their notations have a well defined semantics. Formula manipulations can be applied to further implementation, to check correctness and/or to derive test cases [Bei95].

[1] However, that does not mean that a semantic cannot be defined.

Formal methods have been applied in various fields of application (this includes security-critical software systems as well as embedded systems or hardware systems). There are countries where laws are mandating formal methods (e.g. the UK defense standard for such systems, DefStd. 00-55). Institutions like the US Department of Defense or NASA enforce the use of formal methods in their projects in order to ensure high quality of the resulting software product. The already in Chap. 1.2 mentioned CDIS project, carried out in the early 1990s by Praxis, is one of the most widely known and discussed. At Praxis, formal methods also proved their benefits during the code and test stages in the C130J project in the mid 1990s (a project analyzing avionics software for Lockheed [Hal96]).

What remains is the important question about the "real" benefits when using formal methods.

- According to Pfleeger and Hatton [PH97], the CDIS project demonstrates that formal design (in combination with other techniques) yields a *highly reliable* code. Furthermore, it shows that formal methods are very effective in acting as a catalyst for *testing*. The CDIS project is not the only source for that observation. In 1996 Clarke and Wing [CW96] presented a survey of the use of formal specifications, verification and existing tools. More than 120 references are provided, giving the reader a sense of acceptability of formal methods.

- A recently published empirical study from Sobel and Clarkson [SC02] shows that the application of formal analysis provides great benefit to the implementation with respect to *completeness*. The most important result of the study is that the group using formal methods nearly passed 100% of the standard set of test cases in comparison to 45.5% passed by the control teams.

- Another study, conducted by Samson, Nevill and Dugard in 1987 [SND87], shows that there is a *strong and direct influence* of specification metrics onto metrics of the implementation. Cost estimates can be refined by attributes based on specification metrics. If, for example, a function point analysis leads to a first estimation, a further estimation can be given according to specification attributes. By using specification metrics, an estimate is possible at a much earlier stage in the development process.

2.2 Controversy

It looks like a silver-bullet, but a debate about formal methods began already at the beginning of the 1990s. On the one side there have been people saying that formal methods are essential in order to guarantee quality. On the other side people

have been arguing that formal methods are too expensive. These were arguments starting from two different premises: the first one focused on gaining assurance and the second one focused on cost. Nevertheless, with the application of formal methods up to the mid 1990s, both arguments began to vanish. In a collection of articles concerning formal methods Anthony Hall [HDR+96, p.22] writes that

> it is possible to produce software, even critical software, without formal methods; we also know that it is horrible expensive. What is only recently becoming clear is that it is practical to produce software, even noncritical software, using formal methods; it is also, as far as we can tell, cheaper to do it that way.

It seems to be untypical that the above statement is not more detailed when talking about cost. In the same article Anthony Hall also admits that it is difficult to be more precise. When analyzing specifications' impediments, five statements of criticism (abbreviated by C1 to C5) can be quoted. The impreciseness quoted above becomes the first statement of criticism.

C1 Up to now there is no measurement system available for formal methods.

With this observation it becomes clear why positive effects of formal methods are always shown via anecdotal evidence only. Fenton and Kaposi [FK89, p.7] put it another way:

> In the absence of a suitable measurement system, there is no chance of validating the claims of the formal methods community that their models and theories enhance the quality of software products and improve the cost-effectiveness of software processes.

However, it is not only the absence of suitable metrics. Software engineers still shy at formal methods. Whether it is a myth and/or a fact, formal specifications are still criticized to be hard to understand. The well known criticism out of Hall's seven myths of formal methods [Hal90] and Bowen's seven more myths [BH95] can be summarized as follows (thus, adding further arguments to the criticism quoted above):

C2 Mathematical constructs and operators are requiring expertise in mathematics and formal notations.

C3 There is a lacking connection to other representational forms, regardless of upstream or downstream in the software development process. Specification expertise and expertise in requirements analysis is needed.

C4 When specifications are getting large, they are again containing too much information. Hence the reader is overwhelmed due to the semantic compactness of the specification.

It is uncontested (criticism C2 and C3) that some kind of expertise is needed for applying formal methods successfully; however, criticism C4 points out that specifications are often too complex to be understandable. The situation is even worse, as there are almost no tools (of production-quality) available. Fact is that quite a lot of tools have been built by the academic community [CW96], but most of them are for research, rather than for software development. They still have to be improved with regard to robustness and performance (as quoted by Holoway and Butler in [HDR+96, p.25]). This becomes the fifth statement of criticism:

C5 Only inadequate tools are available for formal methods. It takes great effort to learn how to use them effectively, be it that they use specialized notations from mathematical logic, or be it that they are still not robust enough for being used during software development.

The good news is that this criticism will vanish in the near future as re-write systems are getting more powerful and companies (like IFAD) are providing tool-sets (e.g. the VDM toolkit[2]) for integrating formal methods in the software development process.

2.3 Dealing with Myths

Up to now examples of the successful use of formal methods have been elaborated; five statements of criticism (missing metrics, paradigm and specification expertise, complexity and missing tools) have been identified. However, they have been left hanging in the air, so far. Despite criticisms C2 to C5 it seems uncontested (following the arguments and sources given in [CW96]) that formal methods *can* pay off if they are combined with other techniques and formal specifications *can* significantly improve the quality of the overall system. The question is how to ensure that they really do?

Before we devote ourselves back to this question, another important fact about complexity has to be taken into account: complexity exerts an immense influence on the quality of the system.

An empirical study conducted by Vinter, Loomes and Kornbrot in 1998 showed that even specifications contain errors [VLK98] (be it requirement errors or be it mathematical constructs that are used in a wrong or in an incomprehensible way).

[2] See http://www.ifad.dk/Products/products.htm. Last visited: Sep. 2003.

As is known from software development, errors propagate in the software development process stages, whereas the cost of finding and repairing them raises exponentially [Hum89, p.364]. To make use of all the merits of formal specifications, it is essential to avoid or at least identify errors or misinterpretations as early as possible – in other words, the specification has to fulfill the requirements of *high quality specifications*.

This revives the criticisms raised above. Paradigm and specification expertise will, of course, still be required and there is good hope that these arguments of criticism (C2 and C3) will vanish in the future as more and more curricula are offering courses dealing with formal methods. Remain the other arguments.

Complexity has to be reduced, or at least methods have to be provided in order to support readers of formal specifications. Furthermore, expressive metrics have to be defined in order to support different types of estimations during the software developing process.

2.4 Metrics

In the above section the terms "metrics" and "quality" have only been mentioned but not explained in further detail.

A metric is a size for the measurement of a certain characteristic of an artifact, be it a program, a module or a specification. Generally speaking, a *software (or specification) metric* describes a measurement that has a distinct relationship to a software system, to the software development process or to the documentation. This measurement characterizes numerically determined characteristics of the software which presupposes that a clear definition of the software characteristics is available. In other words, it must be clearly recognizable that such a system or artifact fulfills a characteristic to a greater extent than another object.

A pragmatic organization for the classification of metrics is the separation into linguistic metrics and structural metrics [Bei95], whereby combinations are quite possible. Linguistic metrics only regard the program (literals) without a deeper interpretation of the text itself. A typical metrics of this type is the number of lines of code (LOC) going back to Boehm [Boe81]. Another metric of this class is that of Halstead [Hal77, p.9], who suggests to take the number of different operands and operators into account. Based on these basic types of metrics, a metric for the difficulty/complexity of the program is calculated (see Chap. 6.1.2 for details).

Structural metrics usually refer to the relations between the individual sections of a program. A well-known metric is that of cyclomatic complexity $v(G)$ going back to McCabe [McC76]. This metric is based on the number of edges, nodes and the number of connected components in the flow-graph. Generally speaking, structural metrics are typically based on control and/or data flow relationships.

It is conceivable to apply linguistic metrics to specifications (the work of Samson, Nevill and Dugard [SND87] already relies on these metrics) - the number of predicates in Z schemata or VDM modules, for example, can easily be determined. However, applying structural metrics is more difficult, as, due to the declarative nature of formal specification languages, the basis for the calculation (explicit control- and data-flow) is, in most cases, missing.

2.5 Quality

Quality turned out to be one of the key factors for the success or failure of a product. In fact, the meaning of the term "quality" depends on the perspective from which individual users look upon it. System testers with the emphasis on requirements, for example, have different viewpoints on quality than end users with the emphasis on the usefulness of the product. Overall, quality also means the absence of defects that would cause a system to behave unpredictable or stop successful execution [Jon78]. When speaking about quality in general, we usually focus on *software quality*, but quality considerations are also relevant for specifications[3].

The ISO norm 8402-1994 defines "quality" as follows:

> Quality is the totality of features and characteristics of a product that bear on its ability to satisfy stated or implied needs.

In many cases the pure use of individual quality criteria is not sufficient and therefore quality models have been developed. There are several so called FCM (Factor-Criteria-Metrics) models available [BKP98a, p.258ff], but in all cases quality is described by means of quality factors (like usability) and their related sub-factors (like comprehensibility and learn-ability). Furthermore, there are process models which lead to an enterprise- and/or development- specific quality model. An example is the GQM (Goal-Question-Metric) model from Basili and Rombach [RB87]. They suggest six steps in order to identify product and/or process specific quality factors.

A large number of existing quality factors goes back to attributes introduced by McCall et.al in 1977 [MRW77]. McCall differentiates between two levels of attributes: on the one hand there are quality factors that are not directly measurable, and on the other hand there are quality criteria which are either subjectively or objectively measurable. In McCall's system it is possible to combine quality criteria in order to reason about quality factors. An example for a quality factor is the reliability of a system. Reliability is not measurable but one can reason about reliability if the number of errors so far found in the system is known.

[3] According to Boehm [Boe80] more than 60% of all defects in a software system are introduced before the beginning of the coding phase.

A further, common distinction is the one between internal and external attributes. Internal attributes (like size and error) can be measured directly on the basis of the product; external attributes (like usefulness and maintenance) depend on the application of the system. External attributes can only indirectly be measured and correspond to McCall's quality factors. Frequently mentioned external attributes are correctness, reliability, usefulness, comprehensibility, maintenance and reusability.

When dealing with the quality of specifications (from the perspective of forward engineering), the absence of defects is relevant. But seen from the perspective of reverse engineering or maintenance, external attributes count. If at all and how these metrics, coming from "traditional" software development, fit in the area of specifications will be explored in the following section.

2.6 Specifications' Metrics and Quality

In the field of software development it is common to use quality indicators and metrics. It is also common to speak about the quality of software, the quality of a document and the quality of processes, but rather rarely one speaks (or even thinks) about the quality of specifications. In fact, measures are hard to find but if one uses measures, only linguistic metrics are used. They are described in more detail in Chap. 6.

A typical linguistic metric has been proposed by Vinter, Loomes and Kornbrot [VLK98]: they are using the number of lines of specification code and the number of operators. Another metric it that of Kokol [KPHR99], where the idea of the so-called α-metrics (which is based on the measurement of information content and entropy) is extended to formal specifications. As mentioned earlier, it is sometimes easier to compare. Jilani, Desharnais and Mili [JDM01] present an approach of applying measures of distance between specifications (functional distance is calculated by using similarity properties of underlying graphs) – however, with that measure they address a different complex of problems: that of software retrieval and reuse – they do not address formal specification quality.

When using formal specifications all quality factors mentioned above (like "correctness" and "reliability") seem to be fulfilled automatically. When taking a closer look onto the problem it becomes obvious that like with conventional programs, errors can be introduced. The study of Vinter, Loomes and Kornbrot [VLK98] demonstrates that the number of defects to be found in formal specifications strongly correlates with the number of implications used in predicates of a Z-specification. It also correlates with the perceived complexity of the specification itself. This is a strong evidence that specifications' quality factors are related to specifications' quality attributes.

Talking about the quality of specifications requires knowledge about several specifications' characteristics. For that reason it is useful to search for attributes that permit the identification of quality factors. To be more explicit, we are interested in

- making predictions about the correctness and reliability of a specification,
- making predictions about the expenditure (the costs) of a possible implementation and
- making predictions about the complexity of the specification.

For the first and second statement quantification can be conducted by calculating internal attributes. For the first one it is feasible to use attributes like the number of specific (logical) operators in the specification and some variant of Halstead's metrics. The second prediction can be based on the same attributes, eventually extended by the total number of lines of specification code, the number of predicates or the number of specification objects (if the formal notation, as in Z, supports this type of abstraction). For the description of complexity, however, external attributes are necessary which will be inspected in more detail hereafter.

All factors defined in the field of the traditional software development process seem to be meaningful and useful in the field of specifications. The calculation of these factors, however, is different to calculating programming factors. As we will see in Chap. 3, explicit control- and data flow cannot be assumed when dealing with formal specifications. All too often these clues for partial comprehension are only implicitly available which further increases the complexity of specifications.

2.7 Inherent Complexity of Specifications

Large specifications may contain several thousand lines of specification text (the deducted software system might consist of millions of lines of code). Thus, the set of overwhelming details of a specification often cannot be understood by a single person anymore. We are used to say: things are getting complex!

Up to the middle of the 20^{th} century the word "complexity" appeared to be merely an antonym of "simplicity" [Edm99, p.17], but in fact, complexity is often not so much measured as it is compared. We are talking about *relative* complexity, without an appropriate framework, however, any judgment of complexity would be meaningless. But what could that framework look like?

In 1979 Curtis et. al. [CSM+79] defined the notion of psychological complexity to be based on the number of statements. Another way of defining complexity is the roundabout via resources. Basili [Bas80] defines (software) complexity as

> ... a measure of the resources expended by another system while interacting with the piece of software.

In both articles (Curtis' et.al and Basili's) the importance of *structured programming techniques* is pointed out. But it is not only structure that guarantees a predictable, less complex system. Alagar and Periyasami [AP98] quote following types of complexity:

I1 Environmental Complexity. The usage of a system is defined by many constraints (e.g. time-dependency). As the system also effects the environment, the environmental context has to be well understood.

I2 Application Domain Complexity. The software depends on models of real world objects, models that in most cases can only approximate domain objects. The uncertainty about the real nature of those objects finds expression in the application domain complexity.

I3 Communication Complexity. As software systems are growing in size, only a group of people can be given the task of development. Communication (also communication of tacit knowledge) becomes a key factor which influences the expenditure for managing the project.

I4 Structural Complexity, consisting of Management- and Technical Complexity. The hierarchy of the development team and interwoven development activities contribute to the overall management complexity. In addition, the coupling among system's modules and their interrelationship determine the level of technical complexity, too.

I5 Size Complexity. Size can be (and is) experienced at first hand. According to Alagar and Periyasamy it is probably the most critical type of complexity. They cite a study of Leveson [Lev91] arguing that almost all accidents with software involved are due to this kind of complexity.

Once again, size seems to be the most influencing (and limiting) criteria when trying to comprehend a system. But is it only the size? As is argued by Mittermeir and Bollin in [MB03], 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' behavior 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.

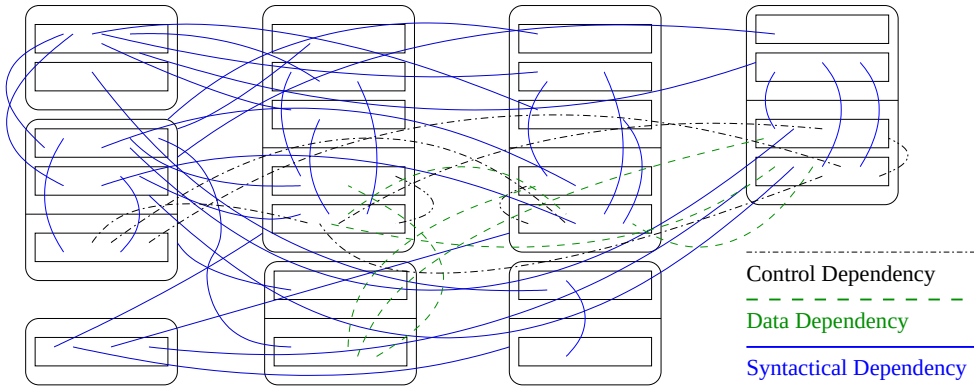


Fig. 2.1: Direct and indirect dependencies in the Z-specification of the Birthday-Book [Spi89b]. The figure demonstrates the huge number of dependencies between parts of the specification. The boxes and the meaning of the dependencies will be explained in detail in Chap. 4.3.2.

Bruce Edmonds introduces the term *analytical complexity* [Edm99, p.86] for this type of difficulty when trying to comprehend expressions. There are several reasons why the reconstruction of the original concepts behind a formal specification is that hard. Reasons for analytical complexity are:

I6 Missing Redundancy. The density of expressions has an important impact on comprehension activities. Whilst humans are used to listen and talk with equal ease in their natural language, the situation is totally different when written communication is used. During a conversation a *dialog* emerges, consisting of questions, counter-questions and answers. With written communication we do not have this chance of constant probing. When using formal specifications, this kind of reassurance is missing.

I7 Too few clues for reconstructing the original structure. Putting too much structure into a specification is usually understood to be a hint towards implementation. While this is true (and can be partially avoided) on the detail level of the specification, larger specifications have no built-in methodologies for structuring. Of course, some notations provide abstraction techniques on the granularity level of objects, but they are of no use when specifications are getting really big.

To demonstrate the argument on a trivial example, one might consider the number of interrelationships (see Fig. 2.1) between concepts used in the toy-specification of a Birthday-Book (BB for short) [Spi89b]. The BB specification is a meaningful but small specification and will be used as an experimental object more than once in this work. The full specification text of the BB-

specification can be found in App. C.1. The interesting fact about it is that the specification itself only consists of 8 objects/concepts and 24 lines of specification code – but the number of relations between them is overwhelming. The number of dependencies between the objects (see Fig. 2.1) indicates that even small specifications are rather complex and (definitely) not easy to comprehend.

I8 Too few clues for reconstructing the behavior. At least with small programs the well defined execution sequence among statements allows for partial comprehension. We are capable of obtaining some understanding by performing a desk-check in the form of a program "run" with some assumed values. 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. One no longer has that built-in clue for partial comprehension.

Most of the complexity factors mentioned above (I1 to I4, partly I5) are indeed inherent in the nature of the task and cannot be influenced, reduced or even eliminated. At least suitable process models, communication techniques and motivated as well as skilled team-members help to overcome environmental, application domain, communication, technical and management complexity.

Missing redundancy (I6) is a property of the formal specification language and would imply the use of rewrite-systems when elaborating on the density of expressions. Rewrite-systems can be very useful when dealing with specifications, but they are time-consuming and require special skills. So the complexity of size (I5) and analytical complexity (I6 to I8) still remains. A lot of solutions have been provided in the field of program comprehension which deal with the problem of size or analytical complexity. When adopting these approaches to specifications successfully it will also be possible to handle that types of specification complexity.

2.8 Questions Revisited

Over the last 20 years several arguments have been raised rejecting or supporting formal methods and formal specifications. By and large there are at least two aspects impeding the further success of formal methods during software development: missing (suitable) metrics and complexity. In the section above complexity has been considered from a more analytical point of view and has been divided into the complexity of size, analytical complexity and some kind of application inherent complexity.

Looking back to Chap. 1, the overall objective was to find out whether it is possible to *reduce the complexity* of specifications. This chapter analyzed the term

complexity in the context of specifications and suggested to reduce complexity by dealing with size complexity and analytical complexity. Seen from this angle, the questions to be answered in this work (and raised in Chap. 1.2) can be deepened as follows:

Q1 *What kind of abridgements are practicable?* Imagine that we already reduced the complexity of a specification. This could have happened (i) by reducing the size, (ii) by adding redundant information or (iii) by providing aids for reconstructing structure or behavior. When reducing the size or adding redundancy, one is going to change the specification. It is obvious that it is not practicable to just delete or add lines of specification code. Providing clues for structure or behavior also means to remove either irrelevant portions of specification code or to add necessary information (by making hidden information explicit). In both cases, the type of reduction counts.

Q2 *How can abridgements be achieved?* The answer depends on the type of abridgement that has been chosen. The first possibility is to just *reduce* the size and thus omit (for a specific problem at hand) irrelevant parts of a specification. Not all reductions are useful (see question Q1) – but basically the problem at hand is supported by scaling-down the original formal specification. The second possibility aims at supporting the *reconstruction* of structure and/or behavior of the original specification. So here implicit information is made explicit, thus *adding* clues to the specification. Of course both approaches can be combined.

Q3 *Can a reduction of complexity be expounded?* If complexity was simply defined by size complexity, the answer would be trivial. When dealing with analytical complexity, however, the answer is not that simple. Redundancy could be measured by some sense of the underlying entropy of the specification text - but that would indeed be detrimental to the characteristics of formal specification notations' – the style of expressing thoughts should not be dictated. On the other hand the clues for structure and behavior are (at least partially) available – hidden in the syntax and semantics of the specification language. The point is that these hidden dependencies between parts of the specification describe (or better: constitute) the analytical complexity – so measures expressing this implicit information can be referred to as metrics for analytical complexity.

Answers to the above questions belong to the field of specification comprehension. The above section already mentioned that there are similar problems to be dealt with in the field of programming languages (e.g. size complexity or too complex structures). The following chapter fills the gap between the vastly unexplored field of specification comprehension and program comprehension.

3. INTRODUCTION TO SPECIFICATION COMPREHENSION

To describe is to understand
To program is to describe
Therefore
To program is to understand

Kristen Nygaard [1926 – 2002]

Chap. 2 presented formal specifications in more detail and illustrated popular facts and fallacies. Formal specifications were identified as very complex constructs, and reasons were given for using formal specifications. The chapter closed with the objective of reducing the complexity of specifications and three questions to be answered in this work were raised.

Formal specifications are not the only documents that are difficult to understand. Due to increasing requirements, both, program code and design documents are growing drastically in size. *Program comprehension* is a field of research where various techniques, languages and tools have been developed to account for this problem.

The objective of this chapter is to demonstrate that the application of program comprehension approaches to specifications is possible. It introduces the field of *specification comprehension*. This is done in two stages: firstly, by introducing program comprehension approaches and secondly, by mapping them to the field of specification comprehension. In this chapter quite a lot of time is spent on program comprehension, as for most of its approaches and ideas this transformation is possible and useful. The crux of the matter is to understand the limits. Thus, this chapter also describes the differences between specifications and programs. Based on a detailed discussion of program comprehension and specification comprehension approaches, these limits are worked out, providing the basis for the objectives of the subsequent chapter: the identification of specification abstractions.

3.1 Comprehending Complex Systems

Pondering the famous statement of Kristen Nygaard (quoted above) it is clear that if a person describes something (by using some kind of formal or semiformal notation),

this person is likely to be fully aware of its meaning. In other words, *comprehension* presupposes complete understanding of something. To comprehend means to connect something which is already on an observers mind to a conception model of that something [Mit00]. Comprehending is an important aspect in humans' life and every day life presents several opportunities to train comprehension activities. It should actually not be a problem. But why is the act of comprehending (software) systems still that difficult?

Chap. 2 provided the answer already: besides missing expertise and knowledge it is complexity making things that difficult. According to Brooks [FPB95, p.182]

> software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the statement level). [...] In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound.

Brooks argues that the complexity of software is an *essential* property and not just an accidental one. This essential complexity and its nonlinear increase with size are the reasons for many problems when developing and/or maintaining software.

Following the arguments of Banker, Davis and Slaughter [BDS98], to understand during forward engineering is not without effort, but the situation gets worse if one is changing from the construction phase to maintenance, reverse engineering or design recovery. A cognitive human information-processing task has to be initiated and input information clues have to be interpreted and manipulated. Again, appropriate concepts have to be *rediscovered* and comprehending implies *re-establishing links* between the clues one can observe and concepts already in mind.

3.1.1 Comprehension Clues

Comprehending really complex systems is not necessarily a lost case. During backward engineering (and partly also during forward engineering) there are several expressive inputs available sustaining the comprehension task. Two of them are the software code and the related documentation. Figure 3.1 describes a (Waterfall-like) model of a development process. It points out the basic clues one can rely on when trying to reconstruct the original mental model behind the system under investigation.

During forward engineering several documents are created. All of them describe the requirements (and their implementation) at different levels of abstraction – and from different points of view. Requirements are typically written down in formal notations which themselves form the basis for creating design documents, user documentations and test documents. All these documents then form the basis for the

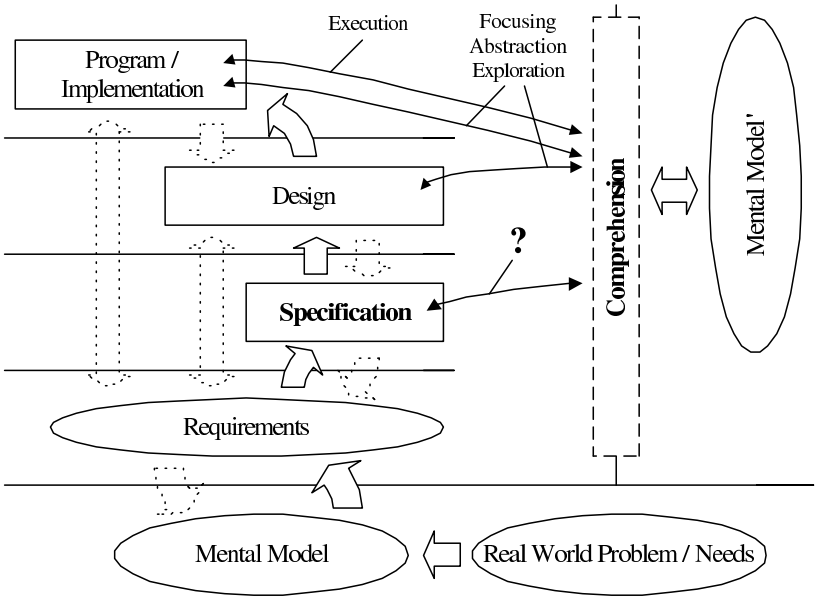


Fig. 3.1: Several sources can be referred to in order to understand a complex system: source code, design documents at different levels of granularity and the formal specification. Whereas there are many comprehension tools and approaches that are based on the program code, support for formal specifications is almost vanishing. This situation is indicated by the question mark between the specification and the comprehension process.

implementation (and test) of the software system. The same holds for other phase-driven development models – as long as documents are used as input sources for subsequent phases.

When trying to understand complex systems there are generally two strategies: firstly, the software system can be executed (a very time-consuming task), and, secondly, the source code and all related documents can be directly used as input for the comprehension process. As shown later in this chapter, focusing, abstraction, and exploration are techniques used to gain knowledge about a system. This cognitive task is often impeded due to missing documents or documents of poor quality. According to Glass [Gla03, p.122] this is one important reason[1] for problems during software comprehension. The better the quality of all documents, the simpler it is to re-establish links between concepts and systems' clues.

This encourages the argument raised in Chap. 2, stating that the quality of all related documents (which includes the formal specification) becomes an important factor that influences the performance of the comprehension process. In fact,

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

documentation, if existent and of good quality, speeds up comprehending complex systems quite a lot. Banker, Davis and Slaughter 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 [BDS98, p.435].

3.1.2 Specifications

When looking at Fig. 3.1 again, one can see that there are many possibilities to gain understanding of the underlying system. So, why are formal specifications still valuable for comprehending complex systems? The program code exactly describes the behavior of the *implemented* system. Thus, one argument could be that taking the underlying software code as the input for the comprehension process is sufficient. In fact, there are 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 various evolutionary steps taking place in the course of system development.

- Even if specifications are getting large, they are smaller than the program that is implementing the described requirements.

It pays off to use formal specifications as a basis for maintenance and reverse-engineering operations (as long as specifications are kept up to date). As Fig. 3.1 points out, up to now there are several approaches supporting program comprehension, but hardly any approaches supporting specification comprehension. One of the objectives of this thesis is to eliminate the question mark in the center of the Fig. 3.1.

3.2 Program Comprehension

This chapter explains terminology, argues why program comprehension is that hard and describes approaches and tools sustaining the engineers in comprehending software systems.

3.2.1 Motivation and Terminology

The lion's share of all the effort that is put into software development is spent on maintaining existing systems. Thus, it is worth improving the situation. There are many estimates on the proportions of resources, but the common opinion is that it ranges between 40% and 80% [Gla03, p.115] – which is quite a lot. As cited in [Rug95], Fjeldstad and Hamlen report

> that 47% and 62% of time spent on actual enhancement and correction tasks are devoted to comprehension activities involving reading the documentation, scanning the source code, and understanding the changes to be made.

Rajlich and Wilde [RW02] bring it to the point: software that is not comprehended cannot be changed. Maintenance is not the only activity that is mainly based on the understanding of the underlying system. Understanding is always essential and the term *program comprehension* describes this activity as

> the process of acquiring knowledge about a software system.

Program comprehension is a rather general term. There are several more specific terms describing activities related to program comprehension [Rug95, p.2]:

- *Reverse Engineering.* It is a process analyzing the software system in order to identify the system's components and their interrelationships. Documents at different levels of abstraction are created.
- *Design Recovery.* It is closely related to reverse engineering. Due to design recovery, already existing documents (e.g. design documents and specifications) are taken into account – in order to gain knowledge about the system.
- *Reengineering.* Reverse engineering moves from the code level to higher levels of abstraction. Reengineering works both ways round – it uses the accumulated knowledge about the system and re-implements the system in a different form.
- *Restructuring.* This process can be seen as reengineering on a smaller scale. The objective of this activity is to re-implement parts of the system (changing local structures) without raising the level of abstraction.

All of the above activities are based on comprehending the whole or at least parts of the underlying system. Storey et.al [SFM99] collected various factors influencing the comprehension process and suggested to classify them by the following three characteristics:

1. *Programmer characteristics.* They include the application domain knowledge, the programming domain knowledge, expertise, creativity, program familiarity and the tool expertise.

2. *Program characteristics.* They consist of the application domain, the programming domain, program size, complexity and the documentation.

3. *Task characteristics.* The differences are expressed by the task type, size and complexity, timing constraints and other environmental factors.

It is remarkable that, once again, the factors size and complexity are not to be neglected. The remainder of this section presents strategies and techniques sustaining the comprehension process. It focuses on abstractions dealing with the issues of size and complexity.

3.2.2 Comprehension Theory

Several models have been built in order to understand how programmers comprehend programs. The so called *mental model* describes the programmers' mental representation of a program or a system to be understood. The *cognitive model* describes the information structures and processes used to form that mental model.

For many years studies observed how programmers understand programs. As a convergence cognitive models for program comprehension have been proposed. The point is that all models have one thing in common: they use existing knowledge to acquire new knowledge and to create a mental representation of the code. According to Mayrhauser [vMV94]) there are so-called *lower-level comprehension models* (or basic models) and *higher-level comprehension models* (or composite models). Lower-level models (in the sequel called M1) and higher-level models (in the sequel called class M2) can be divided into several subclasses.

M1 Lower-level comprehension models:

- Top-down comprehension models. Here, comprehension begins with a high-level goal. The process of understanding starts with a hypothesis concerning the whole piece of code. These goals are then refined in hierarchical fashion forming sub-goals. Examples of top-down models are the Brooks model [Bro78] and the model of Soloway and Ehrlich [SBE83]. Top-down program understanding models are usually applied when the type of code is familiar to the reader.

- Bottom-up comprehension models. With these models of comprehension the understanding is being built bottom up by reading the source code and then

grouping parts[2] of the program to higher level abstractions. Typical bottom-up models are Shneiderman and Mayer's cognitive model [vMV94, p.10] and the Pennington model [vMV94, p.16].

M2 Higher-level comprehension models:

- Knowledge-based understanding model (also known as the Letovsky Model [vMV94, p.9]). This model is a high level cognitive model of program comprehension, based on a knowledge base, a mental model and an assimilation process. The knowledge base itself consists of the programmers' application and expertise, the problem domain knowledge, the rules of discourse, the plans and the goals. The mental model represents the programmer's understanding of the program. The assimilation process describes how the mental model changes by combining the knowledge base and the program information using either bottom-up or top-down strategies.

- Systematic and "as-needed" comprehension models. Programmers either read the code in detail, tracing the control- and data flow abstractions or they take an as-needed approach, focusing only on the code related to the specific task. The model of Soloway [SBE83] combines systematic strategies, "as-needed" strategies and the model of Letovsky into a single model consisting of micro strategies (using read, question, and conjecture and search cycles) and macro strategies (systematic strategies for tracing the flow of the program including simulations and as-needed strategies).

- Integrated approaches. Von Mayrhauser and Vans combined the top-down, bottom up and knowledge based approaches into a single meta-model [vMV94, p.18]. Here, comprehension is obtained by concurrently using different comprehension strategies, even at different levels of abstraction.

Each model represents slightly different strategies for program comprehension. Nevertheless, they are not totally different from each other. All models use matching processes between the (already known) knowledge structures and the program under study. Programs are nothing else than documents written in some formal notation, describing the behavior of the software system.

The strategies mentioned above are not strictly bound to program comprehension. As formal specifications are also documents exactly describing the behavior of the underlying system, the same strategies hold for specification comprehension.

[2] In literature this parts are also sometimes called "chunks".

3.2.3 Comprehension Strategies

In order to gain understanding of a system one will have to analyze the underlying program and/or documentation. This can be done by hand or partly supported by tools. According to Tilley [Til95], three basic activities are typical for the comprehension processes (which, in consequence, also set the requirements for related tools):

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

A2 Knowledge organization by *focusing* and *creating abstractions* for efficient storage and retrieval.

A3 Information *exploration and visualization* using navigation aids, analysis tools and different types of presentation.

All of the above activities are supported by tools, but only activities in class A3 have a real focus on visualization techniques. Activities in class A1 are primarily based on *source-code level comprehension* approaches and serve, in most cases, as a basis for activities in classes A2 and A3. Activities in class A2 involve heavy static and dynamic analysis, but help in the formation of digestive bits/parts of the underlying system.

3.2.4 Data Gathering

The simplest activity at the level of source-code comprehension (activity class A1 in Chap. 3.2.3) is *textual, lexical* and *syntactic analysis* [ASU86]. The grouping and the meaning of characters is important, the syntax and structure of the program guide the reader. Comments are essential and help with forming a first idea of the piece of code. Based on these low-level activities, higher level activities can be started. It is possible to detach specific information and to abstract from too many details.

This is done by looking at the program's *control flow* and *data flow*. The abstract syntax tree resulting from static analysis then forms the basis for constructing a control-flow and data-flow graph, or immediately leads to a *program dependence graph* [FOW87].

Up to now, all approaches are based on static analysis of the program code. *Dynamic analysis* can be used to gain insight in the program under investigation. It includes partial or full *execution*. *Abstract interpretation* [CC77, CC99] is located somewhere in between static and dynamic analysis. Here denotational semantics is used in order to express semantic domains, enabling an abstract re-interpretation of statements.

3.2.5 Knowledge Organisation

A typical strategy is that of looking for *programming abstractions* (activity class A2 in Chap. 3.2.3). In many cases it is important to restrict analysis to parts of the program by neglecting some (for that moment irrelevant) aspects of it. Two popular techniques called *slicing* [Wei79] and *chunking* [BRS+97] serve as a basis for this activity. A third activity is that of looking for programming *patterns* [RW90] or *clichés* [BF99]. Slicing, chunking and the identification of patterns or clichés provide quite a lot of abstraction power. Thus, these techniques will be described in the sequel of this section.

Programming abstractions help to focus on well-defined portions of code. Depending on the problem at hand, a peruser can be assured that if s/he analyzes the respective portion of code, all that needs to be studied for the problem at hand has been considered. Such portions of code are often identified as slices, chunks and clichés or patterns.

The original idea of slicing goes back to the PhD-thesis of Weiser [Wei79]. He defined a slice s as follows:

Definition 3.1: A *slice* s is a reduced, executable program obtained from a program p by removing statements, such that s replicates part of the behavior of p.

A program slice, as defined by Weiser, is based on static data-flow analysis and allows finding the slice in linear time as the transitive closure of a dependency graph. Following the definitions of Weiser, a slice is defined by a program's subset of statements and control predicates that are dependent on a slicing criterion (the point of interest). The slicing criterion, in general, is a pair consisting of a line-number and a set of variables. The calculated slice must preserve the effect of the original program on these variables at the given line number. Ottenstein and Ottenstein [OO84] restated the problem in terms of a reachability problem in a PDG (program dependence graph) in 1984.

However, there are several definitions and extensions to be found in the literature. A common distinction is that of static and dynamic slices. A static slice makes no assumptions regarding a program's input, whereas a dynamic slice relies on a specific test case – only dependencies that occur in a specific execution path of the program are regarded and taken into the slice.

A slice, as defined by Weiser, is sometimes also called a "backward slice" as it is calculated by starting at the slicing criterion and performing a backward traversal of the program. Bergeretti and Carrè [BC85] introduced the notion of a *forward slice*: a forward slice consists of all statements dependent on the slicing criterion. An overview of different approaches can be found in the survey paper of Tip [Tip94].

Another type of abstraction is that of a programming chunk. Burnstein [BRS+97] defines a chunk in two ways. The first and very general definition states:

Definition 3.2: A *chunk* is a sequence of software instructions that achieve a coherent purpose and can be understood outside of the context in which it is used.

According to Burnstein, *chunks* are syntactic and/or semantic abstractions of text structures within source code. Chunks can be further collected and abstracted (in order to build higher level chunks). A more specific definition is given in [BRS+97] as follows:

Definition 3.3: A *chunk* is either a prime, including all primes contained within it, or a sequence of primes that exist within the same programming scope, where for each pair of primes either one prime is data dependent on the other, or both primes are data dependent on a third prime within the sequence.

This second, alternative, definition provides hints for automatically detecting chunks in a program by detecting and following data dependencies. However, the only important and challenging property of a chunk is that it always has to be understandable – the definition itself does not guarantee that the chunk is executable.

Rich and Wills [RW90] took into account that programmers tend to use the same structures over and over. These repeated structures are called *programming patterns or clichés*. The same concept has been taken up again by Broad and Filer [BF99] who defined a program cliché as follows:

Definition 3.4: A *program cliché* is a basic knowledge unit used by programmers to build and recognize code.

Clichés are *commonly-used* computational structures. Prevalently used are data-structure clichés such as stacks, queues, hash tables and algorithmic clichés such as sorting or binary search. Clichés include fragments of code and, as with chunks, can include further clichés. This indicates that a cliché does not necessarily have to be executable, but the claim for being a rather self-contained unit indicates that syntactic dependencies still play a crucial role.

3.2.6 Exploration and Visualization of Programs

Research shows that cognition models and visualization techniques should not be neglected [Won98]. The third class of activities (activity class A3 in Chap. 3.2.3) relies on program understanding (visualization) tools.

The terms *software visualization* (SV) and *program visualization* (PV) are often misunderstood in the literature as they contain the word "visual" (which is derived

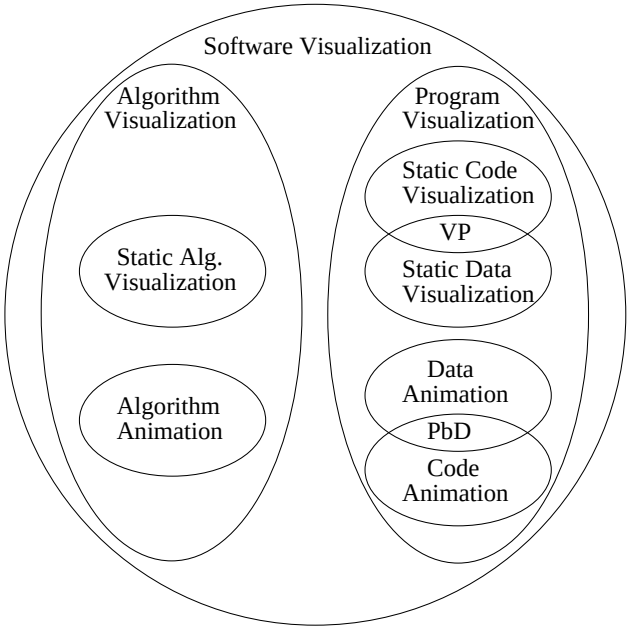


Fig. 3.2: Venn diagram showing the wealth of terms used in the software visualization literature. Please note, that the terms Visual Programming and Programming by Demonstration are not subsets of Program Visualization. Nevertheless, they are related and have a partial overlap. (Out of [SDB+98])

from Latin meaning "sight"). Visualization is not necessarily related only to the displaying and graphical fields [Kad02]. According to the Webster's Encyclopedic Unabridged Dictionary visualization suggests the formation of a mental image. According to [SDB+98], software visualization is the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and the effective use of computer software.

The vast majority of visualization approaches makes use of graphics to support the formation of a mental image of a systems behavior, structure or function. With ongoing improvements in Human Computer Interfaces, visualization through virtual reality systems [SW93] or sound [BH98] becomes possible[3], too.

Specific clues of information are often reconstructed through visualization. Figure 3.2 shows the terms used in the software visualization literature. Up to the late 1980's the term program visualization was used for software visualization. Program visualization, however, refers to the program itself (at a lower level) and not to the underlying algorithm (which is a design at a higher level). Program visualization shows the actual program code or data structures either statically or dynamically,

[3] Visualization through sound is also called "program auralization".

whereas algorithm visualization is a kind of high level description of software (which can be static or animated).

As an example, *static code* visualization includes some kind of program map or pretty printing of the program code, whereas *static data* visualization might represent the code as a boxes-and-arrows diagram. In our example *data animation* then shows the same diagram, but the content of the boxes and the arrows are changing during the execution of the program. A form of *code animation* could be the highlighting of lines of code during the execution phase. Flowcharts are also simple examples of *static algorithm visualization*. The use of animation techniques (to show how the algorithm is working) then leads to the so-called *dynamic algorithm* visualization.

The terms Visual Programming (VP) and Programming by Demonstration (PbD) are mentioned in Figure 3.2, as there is a relationship between these areas and program visualization. VP is a technique using "visual" (graphical) techniques for the specification of a program. The focus of VP is to make programs easier to write. Program visualization aims at making programs easier to understand.

PbD (sometimes also called Programming by Example) is based on the idea of using user demonstrated examples. Users need not have advanced programming skills, they only have to demonstrate an example and the system infers a program automatically.

3.2.7 Program Comprehension Tools

There are many tools supporting program comprehension. These program comprehension tools can be divided into three basic classes:

T1 Software visualization tools that focus on the textual representation. They mainly use pretty-printing (or maps) for increasing the comprehension process. This class of tools mainly deals with static code and data visualization.

T2 Tools presenting the dynamic execution of programs. They are used for debugging, profiling and for the understanding of run-time behavior. Tools of this class are predominantly used for data and code animation.

T3 Tools for the animation of algorithms and data structures. These tools are the so-called "classical" algorithm visualization tools.

Most approaches only focus on a specific problem domain and do not support different cognition models. Thus, they can only be applied in very specific fields: there are visualizing tools for concurrent programs (PVaniM, [SDB+98, p.237f]), tools in the field of education (beginning with BALSA or Piper, see [SDB+98, p.383f])

or tools in the field of software engineering (e.g. maintaining large systems using SeeSoft [SDB+98, p.315f]).

The above mentioned models have been used to improve and evaluate the RIGI tool [TMO92, MTO+92, SWH98] which has been designed for the support of reverse engineering. The RIGI Reverse Engineering tool uses views to direct the user's focus on visual data and to guide the exploration of spatial data to support program documentation and understanding. In 1993 the RIGI system already consisted of a parsing subsystem, a repository and a graph editor. The reverse-engineering methodology is based on subsystem comprehension, views and hypertext layers.

The reverse engineering process involves parsing of the program which results in a graph where nodes represent functions and data-types and arcs represent dependencies among them. In 1996 the RIGI model was extended by SHriMP views [SM96], simple hierarchical multi-perspective views (fish-eye views) of nested graphs. It was shown [SWF+96] that SHriMP views help to enhance comprehension and navigation aspects.

Size is always a problem (especially for graph structured approaches), but different hyper-medial representations (sound or virtual spaces) and different views (like the above mentioned SHriMP views) are of great help. An upcoming approach to overcome the problem of complexity (introduced by Knight and Munro [KM99, YM98]) is the use of three-dimensional visualization techniques.

3.3 Program Comprehension versus Specification Comprehension

Chap. 2 and Chap. 3.1 discussed the problem of complexity, and Chap. 3.2 presented aids for comprehending complex programs. Formal specifications are, like programs, documents which exactly describe the behavior of a system. Hence it is most likely that approaches for comprehending programs strongly resemble comprehension approaches for specifications. With the background of Chap. 3.2 similarities and differences are analyzed.

3.3.1 Terminology

In analogy to program comprehension, the term "specification comprehension" is defined as follows:

Definition 3.5: *Specification comprehension* is the process of acquiring knowledge about a system's specification.

This definition extends the idea of program comprehension by specifications as the basic input-source to the comprehension processes. It also specifies the general

definition, as the input is limited to specifications only and the processes objective is only to understand the system as described by the specification itself.

In accordance with this definition, program comprehension models (Chap. 3.2.2) and strategies (Chap. 3.2.3) are examined and mapped to the world of specifications.

3.3.2 Comprehension Models

There is no reason to argue that cognitive models describing specification comprehension differ from cognitive models of program comprehension (see Chap. 3.2 and classes M1 and M2). Existing knowledge is used to acquire new knowledge and to create a mental representation of a specification.

M1' Application of lower-level models:

- Top-down comprehension models can be applied without problems. Formal specifications provide a simple and compact description of the system. More important, they provide an abstract view which makes it easier to formulate hypotheses.

- Bottom-up comprehension is also possible, even though it is more difficult to group parts to higher levels of abstractions. This is partly due to the fact that formal specifications are at a quite high level of abstraction already – and further abstraction would be impeded or unnecessary. Otherwise the type, the dialect of the specification language and the structure of the specification affect the ease with which a bottom-up procedure can be used.

M2' Application of higher-level models:

- As lower-level comprehension models are applicable to specifications, higher-level models are also suitable to describe the comprehension process. All models are based on documents. Additional knowledge about the system is gained by the recurring tasks of micro and macro strategies applied onto these documents.

Similarly to program comprehension, the factors influencing the comprehension process are the same, namely, the characteristics of the person trying to comprehend the specification, the characteristics of the specification itself and the characteristics of the task.

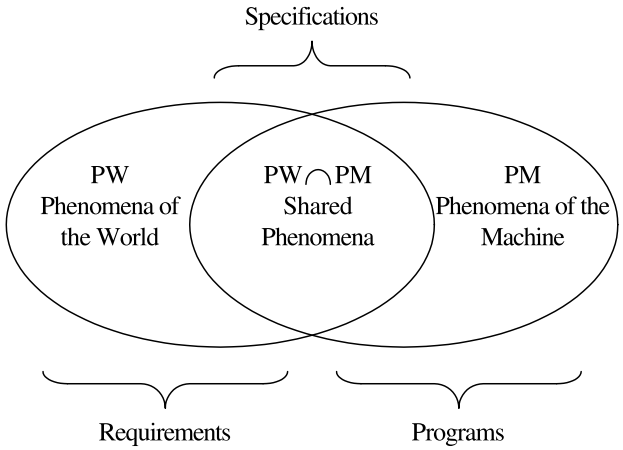


Fig. 3.3: Diagram showing the relationship between requirements, specifications and programs. (The diagram is based on a figure given in [Jac95b].)

3.3.3 Common Features and Differences

Following the software development life-cycle and by moving from requirements to implementation, one is changing the level of abstraction. Michael Jackson [Jac95b] argues that the requirements – the problem – is in the world, and that the solution is the machine which is constructed by software engineers. Requirements, specifications and programs are often closely related, but the relationship is not a simple one[4].

Requirements, on the one hand, are only concerned with the problems and phenomena of the real world. Programs, on the other hand, are concerned with phenomena in the machine (sets PW and PM in Fig. 3.3). According to Jackson, the gap (represented by set $PW \cap PM$) is bridged by specifications, because their purpose is to describe those properties and behavior of the machine which fulfill the requirements of the real world. Thus, a specification is both a requirement and a program [Jac95b, p.285].

Trying to comprehend a system can be considered as concluding from elements in set PM to elements in set PW (by abstracting from machine or implementation related details). Here, specifications are valuable as they exactly deal with the phenomena of the world; but they are also concerned with phenomena of the machine. It is this particular relationship that makes it possible to map program comprehension approaches to specifications.

[4] Michael Jackson mentions four relevant facets: the modelling facet, the interface facet, the engineering facet, and the problem facet.

Despite this common ground the level of abstraction cannot be neglected. Already at the pragmatic level difficulties and differences between requirements, specifications and programs can be observed:

- With specifications it is generally not possible to describe all phenomena of the real world. Additionally, specifications are abstracting from implementation related details – thus, covering only a part of the phenomena of the machine. In many cases the intersection $PW \cap PM$ only forms a small part. Neither a specification nor a program will suffice to reconstruct all truths about the world.

- A specification that describes the world and the machine has to abstract from control properties. Furthermore it is control behavior in the real world that cannot be compared to control-constructs in programs [Jac95b, p.285]. This feature is considered in many (declarative) specification languages. It goes hand in hand with the level of abstraction but constitutes a big difficulty when applying program comprehension approaches like slicing or chunking to specifications.

Specifications are too limited to describe all phenomena of the world. They are not programs, as they are too abstract to be executed. But they are sufficiently related so that program comprehension strategies can be applied to specifications.

3.3.4 Strategies

Programs and formal specifications are documents providing the basis for the comprehension processes. The cognitive models for program and specification comprehension are similar, and even the three basic activities, as described in [Til95], are more or less the same (see Chap. 3.2.3, activity classes A1, A2 and A3). Only on the semantic and syntactic level the differences between specifications and programs turn out to be more important. On the whole it can be said:

A1' The first activity described data gathering using static and dynamic analysis. Source code and specification text comprehension work similarly. Textual, lexical and syntactic analysis is possible and supported by tools[5].

The first big difference pops up when trying to analyze control and data-flow. With imperative programs, an explicit flow of control is given. Line-numbers represent an explicit order among statements or other constructs. Hence, the well-known technique of constructing a PDG can be applied. However,

[5] E.g. cadiz for Z. http://www-users.cs.york.ac.uk/~ian/cadiz. Last visited: Oct. 2003. And vdmsl for VDM. http://www.ifad.dk/Products/vdmtools.htm. Last visited: Oct. 2003.

when looking at declarative specification languages like VDM or Z, there is *no explicit flow of control*. There are also no line numbers and the ordering of the specification's statements is irrelevant. Hence, the notion of data-dependency has to be re-interpreted.

Another difference becomes clear when trying to perform dynamic analysis. Formal specifications are, in general, *not executable*. In some special cases (when predicates are defined in an explicit manner) parts of a specification can automatically be transformed into another language. Partial execution is, even for Z or VDM, with restrictions, feasible. A full execution is not possible[6].

A2' The second activity is based on tasks defined in A1' and focuses on different types of abstractions. The principles of slices, chunks and clichés are well understood for programs and the concepts are useful for focusing on and/or separation of documents. Slices help with focusing on relevant parts in the specification text, whereas chunks and clichés manage to abstract from specification structures.

The problem here is that slices, chunks and clichés are not defined for specifications. Another problem is the fact that these forms of abstraction are partly based on dependencies which are actually not existent. Chap. 4 presents an approach to overcome this problem – thus, providing the basis for specification abstractions.

Abstract interpretation, as something in between static and dynamic analysis, is well supported. Specification languages provide means for describing the semantics and rewrite systems allow for further analysis. In order to gain insight into the meaning of parts of the specification, this class of tools provides means for reformulating the specification and for reasoning about it. This is done by deducing properties of the specification and by arguing about it. See [Jon90, BG94] for more details.

A3' The third class of activities is concerned with information visualization and tools. Up to now only tool class T1 (textual analysis) is broadly supported. The reason for this is the simplicity of static code analysis. However, their limit is reached when the analysis of data- and/or control- dependencies gets necessary. Tool class T2 (dynamic execution) is not fully supported. The same is valid for tool class T3. Translation and animation libraries (see Chap. 4.1.1) for subsets of the specification provide a way out of this dilemma.

[6] For more details on Z transformations see e.g. Possum: http://svrc.it.uq.edu.au/Possum. Last visited: Oct. 2003.

3.4 Summary

Chap. 3 provides a short introduction to comprehension models and approaches for program comprehension. It also connects these models and approaches to the field of specification comprehension. Finally it turns out that there are not many differences. The following observations were made:

- The original mental model behind a specification and a program is the same in most cases, especially when concerning those requirements of the world that are recordable.
- Comprehension models can be applied to specifications without change.
- Comprehension strategies (including the activities known from program comprehension) can be assigned to specifications one to one.
- The notion of programming abstractions can be mapped to specifications. But there are limits and impediments.

The impediments depend on the different natures of programs and specifications. Mapping program comprehension approaches to specifications also has its problems:

- Specifications abstract from implementation related details and the notion of control. In many cases they do not provide control structures and therefore dependencies are not easily deduced. This impedes the application of approaches based on these concepts.
- Up to now there is a lack in the definition of specification abstractions. Widely known types of abstraction (e.g. slices and chunks) are not well defined. Thus, they cannot be derived from a full specification automatically.
- Specifications are, in general, not executable. Even a paper-and-pencil run is impeded due to a missing ordering of statements. However, there are a lot of hidden dependencies and comprehension gets more complicated with increasing quantity of the latter.

Chap. 4 examines these impediments. Firstly, it provides a neat definition of specification abstractions. Secondly, it presents an approach for the identification of specification's control and data dependencies.

4. SPECIFICATION ABSTRACTIONS

Ex parvis saepe magnarum
momenta rerum pendent.

Livius XXVII 9,1

Chap. 3 illustrated the importance of specification comprehension, and demonstrated that a mapping of program comprehension approaches to specifications is promising. One main emphasis was on program comprehension concepts, as a shift to specifications is possible and useful. However, there are relevant differences between specifications and programs. These differences impede the generation of specification abstractions when applying ordinary program comprehension algorithms.

This chapter takes a closer look at these impediments. Firstly, it provides a neat definition of specification abstractions and secondly, it presents an approach for the identification of dependencies in Z specifications.

4.1 State-of-the-Art

With the increasing number (and size) of specifications, there is also a growing demand for support and tools. Up to the mid 1990s several research groups (and companies) worked on the question whether and how formal methods pay off in the software development process (see Chap. 2.1). Several notations have been developed and specification languages have been extended in order to fit the modern object-oriented design principles. However, up to now most of the effort has been put into forward engineering activities: to support the creation of formal specifications, to integrate formal specifications into other environments/methods (e.g. the VDM link to Rational Rose[1]) and to generate code from specifications. Maintenance considerations, on the other side, are rare. The rest of this section presents an overview of specification visualization approaches and then moves on to approaches for reverse-engineering and maintenance activities.

[1] For details see: http://www.ifad.dk/Products/VDMTools/rose-vdmpp.htm. Last visited November 2003.

4.1.1 Specification Visualization

Similar to the classification provided for program comprehension approaches (see Chap. 3.2.7), specification visualization tools and approaches can be assigned to the following three classes (SV1, SV2 and SV3) [MBPRR01]:

SV1 This first class comprises tools for *writing*, *reading* and *browsing* specifications (e.g. providing some kind of pretty-printing). Class SV1 deals with the appropriate visualization of the needed information and can be further divided into three subclasses:

a) The first subclass contains tools for information *formatting* and *focussing*. These tools are related to tools for syntax highlighting in source code or to other tools providing some specific points of focus on textual representations. It is well known from source-code comprehension that the geometrical arrangement of statements substantially aids comprehensibility. The same argument applies for the presented amount of information, the specification styles and the respective arrangement of terms. The Emacs-Editor (with its corresponding LaTeX cadiz or VDM styles) is a tool which belongs to this class. Tools for focusing on specification text do not exist, at least not until now.

b) The second subclass is concerned with the proper visualization of specifications, in the sense of transforming textual information into some *graphical representation*. Here, people reason about the formal model in terms of the graphical representation. The RSL specification language and the corresponding editor [Ped00] provide such a feature: syntactical relationships between statements of the specification are presented in a graphical and hierarchical manner.

c) The third subclass contains cross-level tools that allow *tracing* from a specification to the implementation. Such links (e.g. service channels [PMRR98]) can be built into the system in a rather simple way or be dynamically established with carefully built systems. A cross-section of such approaches can be found in the literature on software evolution [BKP98b].

SV2 This class makes use of *animation* as the next step to render formal specifications "digestible". Here, executable specifications (or sub-specifications) are needed or at least a way to generate test data from specifications as a prerequisite [OJP99]. However, when talking about animation of specifications, the terms *interpretation and execution* of the specification are much more appropriate [Utt94].

In the context of specifications, animation allows posing "questions" that can be answered automatically. This can also be seen as a low-cost alternative to formal proofs. There are two approaches of specification animation. (i) The first one is to translate (subsets of) the specification to another language (a functional language or, in most cases, Prolog) and thus makes interpretation possible. (ii) Another solution is the direct interpretation of the specification. The VDM toolkit follows this approach [FL96] and allows for animation of VDM specifications[2].

Possum [HST97, HST98] follows the first approach and animates the specification after transforming it into an interpretable language called SUM (which itself is based on Z). In Possum, SUM is used to produce a model-oriented formal specification. Possum interprets SUM in two ways: as declarations building an internal representation of SUM declarations (and returning *true*, if the declaration is valid) and as queries, where responses are simplifications of SUM terms of an execution model for state machines[3]. In addition to that, schemata can be called by the Possum environment. Queries to a SUM specification are interpreted and finally (simplified) expressions are returned. Possum allows user-defined Tcl/Tk procedures which are plugged into the existing interface to explicitly support animation. These Tcl/TK procedures cannot alter Possum's state directly. They have to issue special queries. The Tcl/TK procedures then parse the responses and manage the output on the display.

Other animation tools are Jaza (an open source Z animator), PiZA (a tool for animation of Z into Prolog) and the B-toolkit. A good overview of these tools can be found in [Bow00, Utt00].

SV3 The third class of specification comprehension tools aims at *rewriting* a specification and/or *reasoning* about it. Here, the objective is certainly to raise comprehensibility by generating a more compact representation or by focusing on specific characteristics. Usually the level of abstraction is changed or raised.

Ergo [Wat01, URN02], and Isabell [KSW96, NPW02] are environments that can be used. Another environment is MOCHA [dA00], providing model checking, trace containment and system execution. However, rewriting and (formally) reasoning about specifications is a cost-, and in many cases, a resource-intensive task.

[2] In addition to that it also enables the translation to C++ and Java.

[3] As an example, the SUM query $\{g : 1..10 \mid g^2 < 50\}$ leads to the simplification $\{1, 2, 3, 4, 5, 6, 7\}$ as a response.

4.1.2 Looking for Partiality

Existing specification visualization approaches support the creation of specifications and help establishing links to other representational forms. There is no explicit support for comprehension activities. However, as argued by Boehm and Sullivan in their roadmap for SW economics [BS00, p.321], "the fundamental goal of all good design and engineering is to create maximal value added for any given investment". Still most of the life-cycle cost is expended in change. Boehm and Sullivan add "that for a system to create value, the cost of an increment should be proportional to the benefits delivered". The cost itself is determined by several factors and the time needed for the change is one of the most important factors.

When dealing with formal specifications the above argument also holds. It is necessary to optimize performance. One solution is to support developers by providing work-spaces that enable more efficient activities. The key idea is that, in the context of the underlying task, only relevant parts of a specification are presented. What is missing is the support of developers (or maintenance personnel) with well-defined types of specification abstractions, namely *partial specifications*.

- A partial specifications is smaller than the original specification, but contains all relevant parts of interest (for the problem at hand).
- If 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.

The informatics literature contains several concepts of partiality, aiming to provide an interested party just the perspective needed for a particular task. The already discussed notions of *slices and chunks* come to mind now. They have been introduced in detail in Chap. 3.2.5. In the literature *specification views* and *multi-dimensional hyperslices* are also mentioned. In the following the applicability of these concepts of partiality to specifications is discussed.

1. *Slices.* Slices have not only been applied to programs. The concept of Weiser [Wei79] has been transferred to specifications in the work of Oda and Araki [OA93] (who first defined a static forward slice) and Chang and Richardson [CR94] (who extended the idea of Oda and Araki by a dynamic slice). However, a direct transfer to specifications is still problematic, since Weiser-slices depend on control-flow which is not existent in Z-specifications. The definitions will be elaborated on in the sequel of Chap. 4.2.2.

2. *Chunks.* Chunks have been introduced by Burnstein [BRS+97], but have never been used for specifications. However, chunks promise a slim and fascinating concept. Specification chunks have first been mentioned in [Bol02] and have been defined in [BM03]. The application of chunks and the exact definition will be provided in Chap. 4.2.2.

3. *Views.* They have initially been defined in the data base area and are introduced to the specification literature by Jackson [Jac95a]. He defined views as partial specifications (consisting of a state space and a set of operations) and put these views together to end up with the full specification. By using multiple representations of different views, it is possible to improve the clarity and modularization of specifications, following the idea of separation of concerns. This is done by composing several views and by linking them through their states and operations. As an illustration of views, Jackson refers to the popular example of a word-processor specification. There text-oriented functions like search/replace could be separated from text justification. Each of these views then has its own state space. The interaction between these states is postponed until the views are composed. As Jackson argues in [Jac95a, p.24],

   > separating different aspects of the function of a system into different views allows each to be expressed in its most natural representation.

   The composition itself is supported by a number of invariants and operations. One should note, though, that database views are drawn from a given conceptual schema. As long as update operations via views are restricted, the view mechanism necessitates no additional integrity constraints on the schema. Jackson's specification views, though, are bottom-up constructions and the full specification of the system apparently has to allow for updates on the state space. Hence, developing software specifications via views requires additional support structures to maintain consistency of all views [MB03].

   Views are definitely partial specifications. A view is smaller than the full specification and focuses on a specific aspect of the functionality. However, a broader application of views is impeded for two reasons:

   (a) Views are especially well-suited for Z. The reason is that Z supports global invariants, allows for conjoining operations and combines pre- and post-conditions with a logical AND operation. This simplifies the combination of states and operations. In fact, these properties are vital for working with views. They are, thus, not that suitable for other specification languages.

(b) Views are bottom-up constructs. They are not designed to partition the specification *after* its completion. If a specification is composed of several views, these views can be reconstructed by reversing (or omitting) the composition operations. If a specification is not view-structured, the separation into views is not straight-forward. Slicing and clustering approaches might decompose the specification in respect to a specific functionality or state variable, but invariants for view composition cannot be deduced this way.

4. *Multi-dimensional hyperslices.* Hyperslices and the related concepts are going back to the work of Tarr et.al [TOHS99] and are introduced in connection with aspect-oriented or subject-oriented programming.

   If done well, the separation of concerns provides many software engineering benefits[4]. However, predominant methodologies enable only orthogonal separation of concerns along a single dimension of composition and decomposition. The approach of Tarr et.al allows simultaneous separation of overlapping concerns in multiple dimensions.

   They define a hyper-space where so-called units are organized in a multi-dimensional matrix. A unit is a syntactic construct which can be a declaration, a class, an interface or a document. Each axis in the matrix represents a dimension in concern and each point on an axis represents a concern in that dimension. The coordinates of a unit indicate all concerns it affects [TOHS99].

   The term "hyperslice" combines the two terms of "program slicing" and "hyper-planes". Slicing envisions the action of cutting through a system, whereas hyper-planes indicate that the concern is encapsulated in a space (across multiple dimensions) defined by the dimensions of concern. Units can be combined to form compound units or modules. As a result, a hyperslice is nothing else than a set of modules written in any formalism.

   However, the approach is not suitable as a concept for generating partial specifications out of a given specification.

   (a) The identification of concerns is not a trivial task and there are no ways to detect to which concern(s) parts of a specification are contributing to.

   (b) The approach is intended to be used in a generative manner. If it is not used during specification development, it cannot be used as a means of generating partial specifications during backward engineering. The hyper-space is missing.

[4] The application of Parnas' principle of separation of concern renders reduced complexity, improves reusability and also allows for simpler evolution.

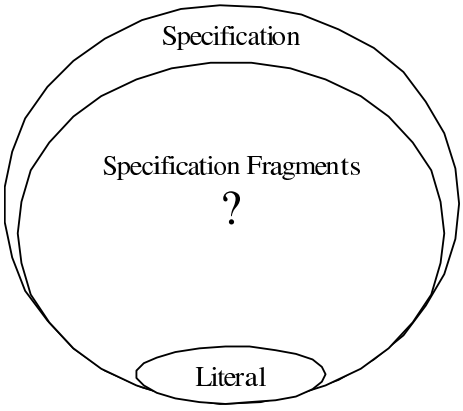


Fig. 4.1: A formal specification consists of a set of specification literals. However, these literals are too fine granular to be understandable outside a given context. For different problems different forms of abstraction are useful. What is needed is a partial specification that is smaller than the original specification, but contains sufficient information to solve the problem at hand. The motivation for Chap. 4.2 is to identify suitable forms of abstraction, in order to fill in the gap accentuated by a question-mark between specification literals and the full specification.

There is definitely a need for partial specifications. Views and hyperslices are mainly generative approaches. They are only useful when they have already been applied during the creation of the specification. Otherwise the concept is of no use. As will be explained in the subsequent section, slices and chunks appear to be promising options. They are decomposing the specification in a well-defined manner. But which parts are relevant?

It becomes necessary to look at possible components of specifications.

4.2 Components of Specifications

For the scope of this work various types of specification abstractions (partial specifications) are interesting. Looking only at terminals (literals) of the specification language is not sufficient. On the one hand those parts of a specification that can be calculated automatically are needed. On the other hand those parts are of interest that can be used to identify (meaningful) patterns in specifications. Fig. 4.1 demonstrates the situation so far.

For the definition of suitable elements a bottom-up approach has been chosen. First, basic (and rudimentary) elements are identified. Then these elements are composed so that they form the basis for higher-level specification concepts.

4.2.1 Syntactic Specification Elements

In general, specifications are constructed from basic (atomic) units. These basic elements are also 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

$$\text{“}\{x : \mathbb{N} \mid x < 5\}''$$

the specification literals are $\{\ '\{',\ 'x',\ ' :',\ '\mathbb{N}',\ ' |',\ ' <',\ '5',\ '\}'\ \}$. However, specification literals are not very expressive when standing alone. It is the combination of literals that makes it rich in content. By aggregating specification literals, prime objects of a specification can be built.

Definition 4.1: 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.

In specification languages these prime objects can be expressions or predicates, but they can also be generic type or schema type definitions. Some examples of Z specification primes are:

$$\text{“}Report ::= OK \mid NOK'',\ \text{“}result! : Report'',\ \textit{or}\ \text{“}[limit : \mathbb{N} \mid limit = 10]''$$

Prime objects are not only 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*. The *Success* operation schema in Z notation (which does nothing else than returning an *OK* when being evaluated)

Success

$$result! : Report$$

$$result! = OK$$

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 [CDHW93].

The important thing is that these prime objects are immutable in the sense that they form the fundamental units (states and operations) on which specifications are built upon. Moreover, it is also important to known that they are merely defined by

syntactical rules of the specification language. 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:

Add

$$name? \notin known$$
$$known' = known \cup \{name? \mapsto date?\}$$

The fragment is an incomplete portion of specification code. It consists of two primes checking and modifying the state. However, for their exact meaning a surrounding text is necessary. Another example would be the Z expression:

$$(Add \land Success) \lor (Delete \land Success)$$

The exact meaning becomes clear when realizing (for the *Add* specification fragment) that *name* is a state variable which contains pairs of names and dates of birth. The second fragment becomes much clearer when bringing to mind that *Add* and *Delete* are operations for adding and removing entries in a birthday book database and that *Success* is used to indicate when the application of the schema operation was successful.

The above fragments are not higher-level prime objects, as they do 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 aid the process of comprehension higher-level specification concepts are also needed – concepts that bear specific types of semantics.

4.2.2 Semantic Specification Concepts

As mentioned above, a partial specification is smaller than the original specification, but it contains all relevant parts of interest for a problem at hand. What is needed is some higher-level abstraction with *clearly defined* semantics. For this reason it

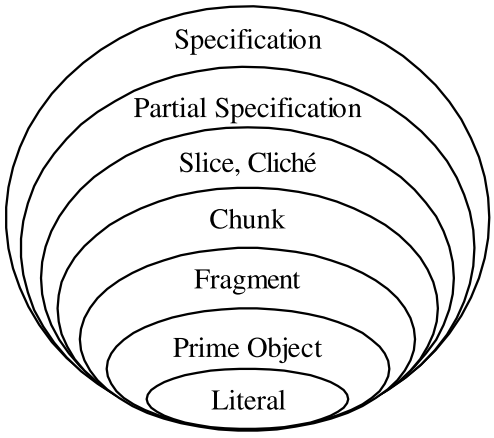


Fig. 4.2: The Venn diagram demonstrates the relationship between basic specification elements (syntactic concepts like literals, primes and fragments) and higher level forms of specification abstractions (semantic concepts like chunks, slices and clichés), latter which contribute to partial specifications.

makes sense to start constructing these abstractions from well-known specification elements, namely from literals and primes.

Based on syntactic elements, several types of abstractions can be derived. Fig. 4.2 presents an overview of those types in form of a Venn diagram. They represent the already known concepts of *chunks*, *slices* and *clichés*.

Chunks, slices and clichés form so-called semantic elements of a specification. They are discussed in more detail in the subsequent sections.

However, when talking about the meaning of an element it is also relevant to know in which context this element can be used, in what parts of a specification the element is valid and from where it can be referred to and used. This introduces the problem of the *scope* of an element. On a very general level, the scope is defined as follows:

Definition 4.2: The **specification scope** of an element denotes those parts of the specification where this element is used or can be referred to.

Scope (and related problems) will be discussed in more detail in Chap. 5.2.2. However, only if the scope of an element is clear it can be used in the correct semantic context.

4.2.3 Specification Chunks

Chunks are syntactic or semantic abstractions of text structures. In accordance with Def. 3.2, to be found in Chap. 3.2.5, 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 4.3: 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 data-dependent on the other or both primes are data-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 [Spi89b]. It is the specification of a simple database called *BB* that allows to store, search for and delete names and dates of birth (see App. C.1 for the full specification in the extended, graphically decorated notation).

$$
\begin{array}{l}
[NAME, DATE] \\
Report ::= OK \mid NOK \\
BB == [known : \mathbb{P}\, NAME;\ birthday : NAME \pfun DATE \mid \\
InitBB == [BB \mid known = \varnothing] \\
Add == [\Delta BB;\ name? : NAME;\ date? : DATE \mid \\
\qquad name? \notin known;\ birthday' = birthday \cup \{name? \mapsto date?\}] \\
Delete == [\Delta BB;\ name? : NAME \mid \\
\qquad name? \in known;\ birthday' = birthday \setminus \{name? \mapsto birthday(name?)\}] \\
Find == [\Xi BB;\ name? : NAME;\ date! : DATE \mid \\
\qquad name? \in known;\ date! = birthday(name?)] \\
Success == [report! : REPORT \mid report! = OK] \\
FunctioningDB == Add \land Delete
\end{array}
$$

Based on the above BB 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 context:

$$
\begin{array}{l}
[NAME, DATE] \\
BB == [known : \mathbb{P}\, NAME;\ birthday : NAME \pfun DATE \mid \\
\qquad known = \mathrm{dom}\, birthday] \\
Add == [\Delta BB;\ name? : NAME;\ date? : DATE \mid \\
\qquad birthday' = birthday \cup \{name? \mapsto date?\}] \\
Delete == [\Delta BB;\ name? : NAME \mid \\
\qquad birthday' = birthday \setminus \{name? \mapsto birthday(name?)\}]
\end{array}
$$

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 Chap. 4.3.

4.2.4 Specification Slices

Program slices are computed by analyzing dependencies in the program source code. However, it is not easy to apply program based slicing approaches to specifications for the following two reasons:

1. In many specifications, control (and data) dependencies are not explicit. In most cases specification languages even do not provide control structures like that of the well-known "*if..then..else*" construct.

2. The slicing criterion (as introduced in Chap. 3.2.5) cannot be mapped from texts containing programming code into texts containing software specifications. A single line is not the dominant item (unit or entity) of a specification.

However, slices have already been defined for specifications. In 1993, Oda and Araki [OA93] first used static slicing techniques for analyzing Z specifications based on a simple definition of data-dependency. One year later, Chang and Richardson [CR94] introduced dynamic specification slicing (by extending the idea of Oda and Araki). By defining a slicing function they indirectly introduced a specification slice:

> Any function removing tokens from the specification can be considered a *slicing function* as long as the specification remains syntactically and semantically correct.

Based on this idea they define a static specification slice $Static_{Dep}$ on a Z specification criterion (s, σ, p, v) where s is a Z specification, σ a schema in s, p is a predicate in σ and v is a variable in p. The result is a set of predicates and enclosing schemas that are dependent on the Z specification criterion. Chang and Richardson state that

> predicates on which p is not dependent are not included in the slice even though their enclosing schemas may be included.

Chang and Richardson define their slicing approach in a top down manner. The limits of their definitions are multi-faceted.

- Their approach is designed for Z-specifications. The slicing criterion is bound to a specific element in a Z specification, namely to one predicate in the declaration section of an operation schema. This criterion has to be restated in order to be applicable to other specification languages.
- Another problem is to define the slicing function as a function that preserves the semantics of the resulting slice. It has to be defined, which elements are to be omitted or included. The approach of Oda and Araki mingles bottom-up and top-down strategies, but fails to deal with the semantics of the included (and omitted) elements.

The above definition of a slicing function is too general and has to be re-stated. On the other hand the definition of the slicing criterion for specifications is too Z-related and thus has to be re-stated, too. Instead of lines, prime objects are taken as basic, structuring units.

Definition 4.4: 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 4.5: 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. 4.5 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:

$$
\begin{array}{l}
[NAME, DATE] \\
BB == [known : \mathbb{P}\, NAME;\ birthday : NAME \pfun DATE \mid \\
InitBB == [BB \mid known = \varnothing] \\
Add == [\Delta BB;\ name? : NAME;\ date? : DATE \mid \\
\qquad name? \notin known;\ birthday' = birthday \cup \{name? \mapsto date?\}] \\
Delete == [\Delta BB;\ name? : NAME \mid \\
\qquad name? \in known;\ birthday' = birthday \setminus \{name? \mapsto birthday(name?)\}] \\
FunctioningDB == Add \land Delete
\end{array}
$$

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.

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. This assures that each slice which is carved out of a specification, has well defined semantics. Starting with the prime representing the slicing criterion, 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) makes 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. This idea of adding well-known elements to an element with clear semantics is also mentioned in [MB03].

4.2.5 Specification Clichés

Another concept leading to partial specifications is that of clichés. As is the case with programs, clichés are basic knowledge units used to build and comprehend specifications. Like Def. 3.4 of a program cliché [BF99], the following definition is provided for specifications:

Definition 4.6: A **specification cliché** is a basic knowledge unit used to build and recognize specification text.

Similar to the definition of a cliché in Chap. 3.2.5, specification clichés might consist of both, fragments of specification code and intermediate specification clichés (from which further specification clichés may be inferred).

Slices and clichés have a lot in common. They represent different concepts, but the differences are rather vanishing. Specification slices are based on the notion of control- and data-dependencies and include all parts of a specification that are somehow syntactically dependent. In contrast to slices, clichés have a heavy focus on concepts including problem specific dependencies.

Slices can be seen as substructures of clichés. The problem here is that clichés cannot be deduced from specifications automatically without additional knowledge. Clichés are recurrent structures. The best way to deal with clichés is to treat them as commonly understood patterns.

Cliché (or pattern) recognition requires a knowledge base with commonly used structures to be compared with. When a sub-structure is identified as a part of a cliché, specification slicing techniques can be used for carving out the cliché from the specification. Thus, clichés might consist of several slices, but a slice is not necessarily a cliché.

At first sight clichés might also be treated as reverse engineering views. The idea behind views is to specify a most suitable specific functionality of the system in a state space. In that sense views examine a specification from different perspectives. The specification then is the result of all solutions' projections to the problem domain. Different perspectives are merged to form the final specification which is the real difference to clichés. On the conceptual level the projection of views to a specification is many-to-one. It is the combination of several perspectives that form the final "image" of the specification. However, a cliché is a one-to-one (or one-to-many) mapping between a (recurring) concept and specification elements. Elements of a cliché thus might be element of a view, but might also belong to several views at a time.

Up to now several forms of abstractions have been discussed. All abstractions are not useful in all situations. Their application depends on the problem at hand and on the point of interest.

4.2.6 The Point of Interest

Chap. 4.2.2 presents several types of abstraction that might be used to support comprehension tasks. It did not suggest when and which type of abstraction should be used. In fact, the answer depends on the problem at hand. Typical situations are as follows:

- *Change of a requirement.* 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 changing a specification prime (or literal). 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 [GZ99]. 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.

At this point the answer to question Q1, raised in Chap. 1.2 ("What kind of abridgements or reductions are practicable for what problem?") can be provided. Almost all tasks of specification comprehension activities can be sustained by slices, chunks and 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 *related*) parts of the specification. With slices, chunks and fragments, the basis for useful abstraction is provided.

Remains the question of how to state the point of interest (the criterion) abstractions are based upon. The answer to this question is split into two parts.

1. Firstly, the focus will have to be set to a specific position in the specification. As argued in Chap. 4.2.2, it only makes sense to look at the smallest entity (with a well defined semantics) available in a specification: the specification *prime*.

2. 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*.

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 controlling dependencies).

4.2.7 Sub-Specifications and Partitions

In addition to the higher-level specification concepts defined above, other forms of partiality are mentioned in literature. They are herein summarized as *sub-specifications* and *specification partitions*. The idea is to identify related (or disjoint) specification fragments, and, in addition to that, to support some form of classification.

A *sub-specification* is a self-contained partial specification. In many cases a sub-specification can be regarded as a slice or as a collection of slices and clichés. Sub-specifications illuminate some, but not all aspects of the original specification. Dealing with specification views [Jac95b] can be seen as dealing with sub-specifications.

On the other hand it is also possible to divide specifications into several parts or partitions. Depending on the algorithms that are used to generate these parts, two other terms for *specification partitions* can be found in literature:

1. *Specification modules.* Carrington, Duke, Hayes and Welsh [CDHW93] defined the notion of a specification module as a set of Z-operations that refer/modify similar sets of state variables.

2. *Specification clusters.* More complex systems can be defined in order to disparate a specification into more comprehensible parts. Clustering algorithms [Wig97] are used to identify related parts. Clustering algorithms are applied to programs, components and software architectures, but up to now they have not been applied to specifications.

The problem with clustering is that clustering methods are concerned with the grouping of entities based on their interrelationships or similarities. On an abstract level, groups of entities are formulated and entities in one group are more closely related to each other than to entities in other groups. But how can similarities between entities of a specification be measured and what are suitable entities?

A step into the direction of clustering and partitioning is the work of Jilani, Desharnais, and Mili [JDM01] who define a measure of functional distance between specifications. With that and the separation of large specifications into smaller partial specifications a clustering algorithm can be formulated. The distances between all partial specifications can now be calculated. Finally, based on a pre-defined trash-hold clusters can be built.

4.3 Deriving Dependencies

Decomposition techniques seem to be very promising approaches for supporting program and specification comprehension. Most of the approaches are based on the generation of a *program dependence graph* (PDG for short). A PDG is a graph where nodes represent statements of the program and edges represent dependence relations [FOW87]. Usually data dependence and control dependence are used. Chap. 3.3.3 named several difficulties in identifying these types of dependencies in specifications. When applying algorithms that are based on a similarly defined *specification dependence graph* (SDG for short), we first need mechanisms for identifying these types of dependencies. And that is the main problem. Especially the notion of control

is not dominant in declarative specifications. Even when an abstract syntax tree is available, control dependencies are not easy to identify. Chap. 5.1 discusses the problems of abstract syntax trees in more detail.

The objective of this section is to identify and to define dependencies in specifications. First well-known dependencies in programs will be looked at (Chap. 4.3.1). Then these dependencies will be mapped to specifications (Chap. 4.3.2).

4.3.1 Dependency Types

Dependencies are used in various fields of application. Apart from program analysis they are also used as a basis for software testability measures [Jun02] as well as for an optimization in concurrent machine performance [Uht91]. Furthermore, dependency based approaches are applied on concurrent logic programs [ZCU97] and even in software architectures [Zha99].

Dependencies can (according to [Uht91]) be divided into three classes: resource, syntactic and semantic dependencies.

- *Resource/Environmental dependencies.* Resource/Environmental dependency arises due to limitations in hardware and service availability. However, they do not become apparent until the execution of the system. They are limitations which are controlled from outside, and they are difficult to detect when the environment is unknown to a large extent.

- *Syntactic dependencies.* Programming languages demand a specific ordering of statements so that the resulting program is kept syntactically correct. The same is true for most specification languages. In fact, if it is not the ordering, at least the existence of specific statements is demanded. In Z, for instance, types have to be defined "before" they can be used.

- *Semantic dependencies.* Semantic dependencies go back to relationships which exist between statements in the program code and require an ordering of instructions. On a very general level a semantic dependency between two statements occurs, when their execution must be ordered for a correct program run. There are two sub-classes of semantic dependencies: data dependency and control (sometimes also called procedural or branch) dependency.

Being limitations determined from outside, resource/environmental dependencies are not treated in the sequel. However, syntactic and semantic dependencies can be identified by looking at the program code at hand. In the sequel syntactic and semantic dependencies are defined for programs. In the next section these dependencies are related to specifications.

Syntactic dependencies are well known from compiler construction. On a very general level, a statement in a program (e.g. "$name = 20;$") syntactically depends on another statement (e.g "$int\ name = 0;$"), if the second statement is needed to keep the program syntactically correct:

Definition 4.7: Syntactic dependencies in programs. Let s_1 and s_2 be two statements of a source program p ($s_1 \neq s_2$). s_1 is said to be *syntactically dependent* on s_2 ($s_1 \rightarrow_s s_2$) if statement s_2 is needed to keep s_1 syntactically correct (in accordance with the language definition).

Control dependencies arise in the program code as a result of decision statements. In general, semantic dependencies are, like syntactic dependencies, unidirectional. This means that statements are control dependent on decision statements, but not the other way round. The result of the execution of the decision statements influences the execution of the statement. Thus, it is important that the decision statement is executed *before* the dependent statement.

Definition 4.8: Control dependencies in programs. Let s_1 and s_2 be two statements of a source program p ($s_1 \neq s_2$). s_1 is said to be *control dependent* on s_2 ($s_1 \rightarrow_c s_2$) if

i. s_2 is a statement representing a conditional predicate and

ii. the result of s_2 determines whether s_1 is executed or not.

Depending on the type of reference to the data element, there are different types of data dependency:

- Flow dependency: A relation between s_1 and s_2 where s_1 writes to a data element of p and s_2 subsequently reads that value.
- Output dependency: A relation between s_1 and s_2 where s_1 writes to a data element and s_2 subsequently overrides it.
- Anti-dependency: A relation between s_1 and s_2 where s_1 uses a data element and s_2 subsequently overrides it.

Anti-dependency, as the name suggests, is a dependency that describes the relationship in an inverse manner. As anti-dependency can be identified by looking for flow dependency, it will not be dealt with in the sequel. The most widely used interpretation is that of flow-dependency upon which the following definition is based.

Definition 4.9: Data dependencies in programs. Let s_1 and s_2 be two statements of a source program p ($s_1 \neq s_2$). s_1 is said to be *data dependent* on s_2 and a data element v ($s_1 \rightarrow_{d,v} s_2$), if

i. s_2 assigns a value to the data element v, and

ii. s_1 refers to the data element v, and

iii. at least one execution path exists from s_2 to s_1 without redefining that data element v.

The next section takes the above definitions of syntactic and semantic dependencies and derives dependencies for formal specifications. However, the identification of these dependencies is not trivial and depends on the specification language at hand. Therefore Chap. 4.4 related the definitions of dependencies to Z. Chap. 5.5 presents an approach for the calculation of syntactic and semantic dependencies in Z specifications.

4.3.2 Specification Dependencies

Semantic dependencies, as described in definitions 4.8 and 4.9, are going back to ordering of instructions. Here one characteristic of specification languages (like Z) is getting cogent: for the most part there is no strict ordering of primes and therefore also no execution order. Nevertheless, a first set of general definitions can be provided.

Definition 4.10: A specification prime p is *syntactical dependent* on a specification prime q, if q is needed to keep p syntactically correct.

Definition 4.11: A specification prime p is *control dependent* on a specification prime q, if q potentially decides whether p is applied or not.

Definition 4.12: 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.

However, the above definitions are too general. They serve as a basis for the identification of dependencies in specifications, but have to be stated more precisely when applied to a specific specification language.

In fact, two terms within the above definitions are important: firstly, the occurrence of a *potential application* of primes and secondly, a series of potential *state changes*. The next section presents an approach for identifying potential applications of primes and state changes in Z.

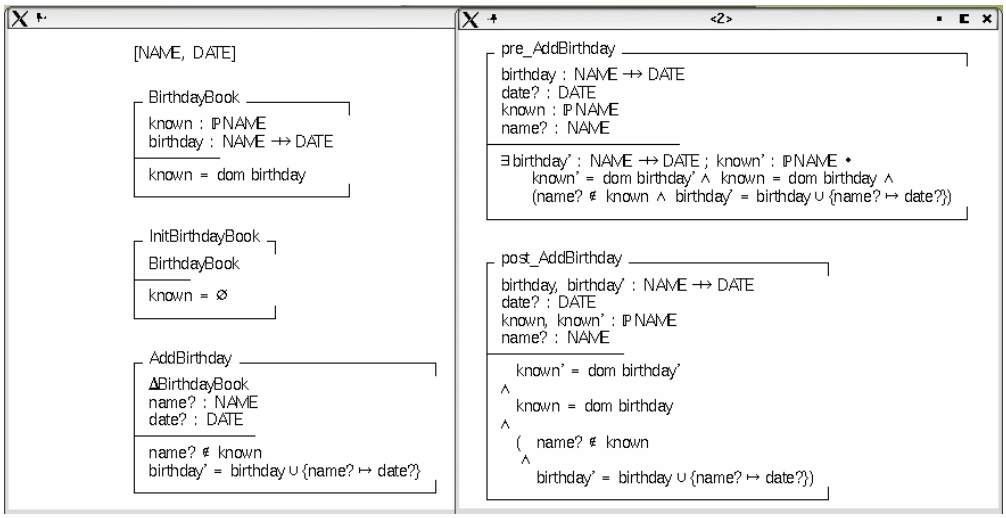


Fig. 4.3: The left window shows the cadiz output of the Z specification of the birthday book (see also Appendix F.1). The second, overlapping window shows the pre- and post-condition calculation performed by cadiz.

4.4 Dependencies in Z

Declarative specification languages do not provide an explicit notion of control. However, the definition of control (Def. 4.8) 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.

Fig. 4.3 presents parts of the birthday book specification (the state space and a schema operation called *AddBirthday* for adding birthday entries to the birthday book). Apart from the definition of two state variables (*name* and *birthday*) the state space *BirthdayBook* contains one predicate (4.1). The schema *AddBirthday* contains two predicates (4.2) and (4.3):

$$known = dom\ birthday \tag{4.1}$$

$$name? \notin known \tag{4.2}$$

$$birthday' = birthday \cup \{name? \mapsto date?\} \tag{4.3}$$

Predicate (4.1) in the *BirthdayBook* state space guarantees that only names that are known are stored in the birthday book database. Predicate (4.2) checks whether the provided name is in the birthday database, whereas predicate (4.3) adds the name and the accompanying date to the database of birthdays. The crux of the

matter is that a semantic analysis of the given operation schema does not generally lead to (4.2) as a pre-condition and (4.3) as a post-condition.

Diller [Dil99] uses the terms *before state invariant* and *pre-condition* for predicates that do not describe an after state. Such a predicate cannot be both a pre-condition and a state invariant. In fact, a state invariant is a pre-condition of *every* operation state. In that sense, predicate (4.1) is a before state invariant and predicate (4.2) is a pre-condition, but not necessarily the pre-condition for all states that could be successfully carried out by the schema operation. An *after state invariant* is a predicate that contains decorated identifiers. The identifier can either be an after state variable (decorated by a $'$), or an output operation (decorated by an $!$).

Z provides a *pre-condition operator* (*pre*) for the calculation of a pre-condition (for those states that can by carried out successfully). It makes a schema out of a schema by hiding all the after and output variables. Fig. 4.3 shows the result of applying the *pre* operator to the *AddBirthday* schema ($pre\_AddBirthday == pre\ AddBirthday$) and presents the predicate which is the pre-condition to the operation schema. The result of the calculation of the post-condition can also be taken from Fig. 4.3 ($post\_AddBirthday == post\ AddBirthday$).

Actually the predicates in *preAddBirthday* and *postAddBirthday* are the true pre-conditions and post-conditions of the operation schema. But as the following example demonstrates, the calculation is not always trivial.

Let *Operation* be an operation schema consisting of a set of declarations and predicates:

Operation

Declaration

Predicate

In Z, the following equation (simple at first sight) defines the pre-condition of the operation schema:

$$pre\ Operation = \exists\, State' \bullet Operation \tag{4.4}$$

The pre-condition of a schema is the collection of before states and inputs for which some after state ($State'$) can be shown to exist. Outputs are hidden by the introduction of an existential quantifier. This means that the following algorithm can be provided for the calculation of the pre-condition (out of [WD96, p.206]):

1. The declaration section *Declaration* is divided into three parts:
   - *Before* contains only inputs and unprimed state components.
   - *After* contains only outputs and primed state components.
   - *Mixed* contains all other declarations and inclusions.
2. If *Mixed* is not empty, every schema in *Mixed* is expanded. All input and before components are added to *Before*. All output and after components are added to *After*. As there may be several levels of schema inclusions, repeat this step until *Mixed* is empty.
3. The pre-condition of *Operation* is then

Operation

$$Before \vdash \exists\, After \bullet Predicate$$

This procedure can be applied to every operation schema, but does not necessarily lead to a simple set of pre-conditions. The following Z-specification is an example of a simple room (with a capacity of *Max* people), where people are allowed to enter. It consists of a state schema *Room* and an operation *Enter* (a *Leave* operation has been omitted for reasons of space). (Supposed that *Person* is a given set, and *Max* is a predefined constant.)

Room

$$in : \mathbb{P}\, Person \vdash \sharp in \leq Max$$

Enter

$$\Delta Room,\ p? : Person \vdash \sharp in < Max,\ p? \notin in,\ in' = in \cup \{p?\}$$

Based on the above step-by-step guide, the pre-condition of *Enter* is calculated. The inclusion has been dissolved, the *before*-components form the new declaration,

and all predicates are enclosed in an existential quantifier expression with the after component as type declaration.

PreEnter

$$d : \mathbb{P}\, Person$$
$$p? : Person$$

$$\exists\, in' : \mathbb{P}\, Person \bullet$$
$$(\sharp in \leq Max \land \sharp in' \leq Max \land$$
$$\sharp in < Max \land$$
$$p? \notin in \land$$
$$in' = in \cup \{p?\})$$

However, when trying to identify pre- or post-condition predicates in Z, this schema paragraph is not that helpful yet. Another step of simplification is necessary. By applying the one-point law[5], the existential quantifier can be removed. This is done by replacing every expression containing "in'" with "$in \cup \{p?\}$". An additional simplification[6] leads to the following schema:

PreEnter

$$d : \mathbb{P}\, Person$$
$$p? : Person$$

$$\sharp in < Max$$
$$p? \notin in$$

That means that the two predicates "$\sharp in < Max$" and "$p? \notin in$" are pre-condition predicates of the *Enter* operation schema.

However, this last step (the simplification) is not straight forward. To avoid time-consuming semantic analysis, another approach has been chooses in this work, namely that of an approximation to the semantic analysis.

4.4.1 Syntactical Approximation to Semantic Analysis

The above section demonstrated that pre-conditions as well as post-conditions can be simplified by applying semantic analysis and rewriting. However, this simplification cannot be automated to its full extend. Rewrite systems *are* necessary and an automatic reduction is costly regarding time and hardware resources. What can

[5] The one-point law is useful for simplifying existential quantifier expressions. The law is defined as follows: $P[t/y] \dashv\vdash \exists\, y : Y \bullet P \land y = t$. Here t is a term of the same type as variable y. y can occur free in the formula P. The law is used to successively replace terms y by corresponding sub-terms t.

[6] As $p \notin in$, we have $\sharp(in \cup \{p\}) = \sharp in + 1$. It holds that $\sharp in < Max$, which implies $\sharp in \leq Max$.

still be observed [Dil99, p.165], is that in most cases the resulting conditions are exactly those predicates that contain only pre-state expressions. They lead to pre-conditions in the specification. The remaining expressions are after state invariants which lead to post-conditions.

Granted that S is an operation schema of a Z specification, it is common to write the schema in a horizontal form as follows:

$$S \;==\; [d;\; d' \mid pr(d);\; po(d, d')]$$

Here, d represents undecorated identifiers and input identifiers. d' represents after-state and output identifiers. pr is a (possible empty) list of predicates containing undecorated and input identifiers. po is a (possible empty) list of predicates containing all types of identifiers. pr thus represents before state invariants and pre-condition predicates. po represents after state invariants. The pre-condition of S can then be written as:

$$\begin{array}{lll} pre\ S & == pre\ [d;\; d' \mid pr(d);\; po(d, d')] & \\ & == [d \mid \exists\, d' \bullet pr(d);\; po(d, d')] & \%\ by\ applying\ equ.\ (4.4) \\ & == [d \mid pr(d);\; \exists\, d' \bullet po(d, d')] & \%\ d\ not\ bound\ in\ \exists\, d' \end{array}$$

This means that $pr(d)$ is an *essential* component of the pre-condition of schema S. When $pr(d)$ is *the* pre-condition of S, it holds that

$$(pre\ S \;==\; [d \mid pr(d)]) \qquad \Leftrightarrow \qquad ([d \mid \exists\, d' \bullet po(d, d')] \;==\; [d \mid true]) \tag{4.5}$$

It only makes sense to calculate pre-conditions if a post state exists. Furthermore, only if the existential quantifier expression can be simplified and eliminated, $pr(d)$ is *the sole* pre-condition.

1. In most cases (according to observations by [Spi89b, WD96, Dil99]) the existential quantifier can be simplified and eliminated. This means that pr is *the* pre-condition of the schema.

2. If the remainder (the existential quantifier expression) cannot be simplified and eliminated, predicate pr is, in any case, part of the pre-condition.

Neglecting the existential expression leads to a weaker pre-condition. In that case it holds

$$pre\ S \;==\; [d \mid pr(d);\; \exists\, d' \bullet po(d, d')] \quad \sqsubseteq \quad pre_w\ S \;==\; [d \mid pr(d)] \tag{4.6}$$

The advantage of weakening the pre-condition as in equation (4.6) is that no semantic analysis has to be performed. On the other hand some kind of inaccuracy is introduced. However, *pr* is part of the pre-condition in any case. There is also a conjunction with the remainder. Thus, *pr* has to be *true* in any case in order to keep the whole pre-condition *true*.

A similar consideration holds for the identification of post-conditions. Every predicate containing output and primed identifiers is treated as a post-condition. In fact, these predicates describe the *after-state* of an operation. Thus, they are definitely elements of the post-condition. However, they do not fully describe a formally deduced post-condition of the schema (see Fig. 4.3).

The basic idea is to skip the calculation of the pre- and post-conditions. Instead, the pre-condition of a schema is said to consist of just the *pr*-predicates (which are before state invariants and pre-condition predicates). The post-condition of a schema is said to consist of just the *po*-predicates (which are after-state invariants). To sum up it can be said that when the remainder is neglected, a *syntactic approximation to the semantic analysis* takes place.

Predicates are one of *the* basic semantic units in our specification. They are called primes in the sequel[7]. For every prime it is possible to determine whether it contains input and unprimed identifers. In that case it is (according to the terminology of Diller) a before-state invariant or pre-condition predicate. Otherwise it contains at least one primed or one output identifier. In that case it is an after-state invariant. The above considerations lead to the following definitions:

Definition 4.13: Z pre-condition prime. A Z-specification prime p is considered a *pre-condition prime*, if prime p is either a before state invariant or a pre-condition predicate.

Definition 4.14: Z post-condition prime. A Z-specification prime p is considered a *post-condition prime*, if prime p is an after state invariant.

Based on these definitions it is possible to identify pre-condition and post-condition primes in Z-specifications. Furthermore, it holds that every prime, representing a predicate in a Z specification, is either a pre-condition prime or a post-condition prime. It cannot be both. The reason is simply that the prime either contains primed or output identifiers (thus, it is an after-state invariant) or the prime is a before state invariant or a pre-condition predicate.

[7] Chap. 5 presents an approach for the identification of primes in specifications. As discussed in more detail in Chap. 5.3.2, predicates form prime objects of Z specifications.

In the birthday book specification (introduced at the beginning of this chapter) predicate (4.3) is a post-condition prime (as the identifier $birthday'$ is decorated denoting an after state invariant). Predicates (4.1) and (4.2) are pre-condition primes. The reason for this is that predicate (4.1) is a before state invariant and predicate (4.2) is a pre-condition predicate.

With this approximation to pre- and post-conditions, relationships between related primes can be identified. This identification is the topic of the subsequent section.

4.4.2 Arrangement of Primes

There is no "sequential execution" of primes in Z specifications. The ordering of primes in the predicate part of a schema operation is irrelevant. Furthermore, it does not matter whether the schema reads like

$$Enter == [\Delta Room;\ p? : Person \mid \sharp in < Max;\ p? \notin in;\ in' = in \cup \{p?\}]$$

or

$$Enter == [p? : Person;\ \Delta Room \mid in' = in \cup \{p?\};\ p? \notin in;\ \sharp in < Max]$$

It is rather a matter of *style*, and several authors suggest a specific format for writing Z specifications (see [Dil99, p.65ff] for a typical style guide).

However, the Z language definition implicitly prescribes some kind of ordering at the syntactical level. This global organization of any Z document is governed by the principle called "definition *before* use" [Dil99, p.64]. This ordering suggests the use of the term "before". In the case of Z it points to the fact that one expression has to be existent in the scope of the other. Some of the rules mentioned by Diller are:

- Z-schemata can only be referred to when they have been defined "before" they are "used". Given set and axiomatic expression definitions are global, but they also have to be defined "before" they can be used.
- In general Z-schema boxes are divided into two parts: a declaration part and a predicate part. Expressions in the schema declaration part describe the scope of the predicate part and are thus defined "before" the predicate part.

This relationship can also be detected in the birthday book example. The given set expression $[NAME,\ DATE]$ has to be defined "before" these sets can be used in the state schema $BirthdayBook$.

Def. 4.7 mentions statements that have to be existent in order to keep other statements syntactically correct. The same holds for Z specifications.

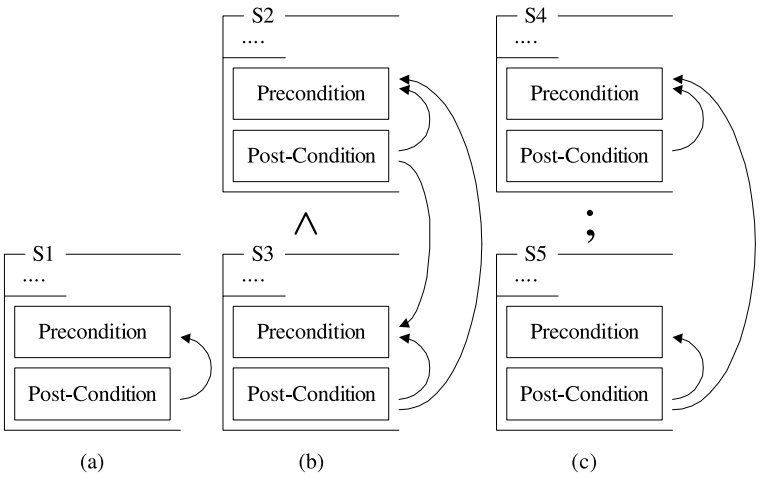


Fig. 4.4: (a) Within a schema a post-condition prime is control dependent on a pre-condition prime. (b) Within a logical combination of two schemata both post-condition primes are control dependent upon both pre-condition primes. (c) The sequential composition of two schemata leads to control dependency between the post-condition of the second schema upon the pre-condition of the first schema, but not vice-versa.

Definition 4.15: Syntactical dependency of Z-primes. A Z-specification prime p is *syntactically dependent* on a specification prime q ($p \neq q$) if

i. p is in the scope of q and

ii. q is needed in order to keep p syntactically correct.

According to Def. 4.11, the notion of control is defined by some kind of "ordering" of primes in the specification. One prime has to be evaluated "before" the other prime is evaluated. The same holds for pre- and post-condition primes. Pre-condition primes are evaluated *before* the evaluation of the post-condition primes. In other words: irrespective of the ordering in the specification text there is the notion of control between a pre- and a post-condition prime (see Fig. 4.4(a)). This leads to the definition of control dependencies between specification primes.

Definition 4.16: Control dependency of Z-primes. A Z-specification prime p is *control dependent* on a specification prime q ($p \neq q$) if

i. q is a pre-condition prime and

ii. p is a post-condition prime in the scope q.

For the identification of dependencies within higher-level primes one has to look for pre- and post-condition primes. These primes are identified according to Def. 4.13 and Def. 4.14. Precondition-predicates are assigned to pre-condition primes. If there is no chance to mix the terms up, they will be used interchangeably in the sequel. However, in order to explicitly denote primes representing predicates of a specification, a retrieve operation Π is defined. Π takes a schema as an argument and returns the set of all pre- and post-condition primes present in the declaration part of the schema.

Definition 4.17: Retrieving primes of Z schemata. Let S be a schema of a syntactically correct specification. Furthermore, let P be the set of primes of S.

The operation Π returns the set of all primes that are representing predicates in schema S. It holds:

$$\Pi\ S\ ==\ \{p : Prime \mid p\ is\ a\ predicate\ prime\ in\ S\}$$

The retrieve operation Π does nothing else than returning the set of primes (representing predicates) to be found in a schema. It does not consider operators when predicates are combined. For example, let S and T be two schemata, and let p_x $(x : 1..4)$ be several predicates:

$$S\ ==\ [d_S;\ d'_S \mid p_1 \vee p_2]$$
$$T\ ==\ [d_T;\ d'_T \mid p_3 \wedge p_4]$$

When Π is applied to S or T only predicate primes are returned. Thus, $\Pi\ S == \{p_1, p_2\}$, and $\Pi\ T == \{p_3, p_4\}$.

As discussed in the sequel, the retrieve function is used to extract primes out of combined schemata. However, first of all dependencies are of interest in a single Z schema. In Z schema boxes, pre- and post-condition primes are in the same scope (and therefore in the scope of each other). According to Def. 4.16, the following definition for the identification of control dependencies within schema boxes can be advanced:

Definition 4.18: 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 $po_S \rightrightarrows_c pr_S$). It holds:

$$po_S \rightrightarrows_c pr_S\ ==\ \forall\, p_1 : po_S;\ p_2 : pr_S \bullet p_1 \rightarrow_c p_2$$

Every predicate which is a pre-condition prime is element of the set pr_S and every predicate which is a post-condition prime is element of the set po_S. Given that S is a schema that consists of two pre-condition primes (pr_1, pr_2) and two post-condition primes (po_1, po_2), the schema can be written as follows:

$$S == [d_S, d'_S \mid pr_1(d_S);\ pr_2(d_S);\ po_1(d_S, d'_S);\ po_2(d_S, d'_S)]$$

Then po_1 is control dependent on pr_1 and pr_2, whereas po_2 is also control dependent on pr_1 and pr_2. For short this can be put down as:

$$(po_1 \cup po_2) \rightrightarrows_c (pr_1 \cup pr_2)$$

Def. 4.18 can be used to provide a rule for the identification of control dependencies in Z schemata.

Rule 4.1: Control dependencies in Z schemata. Let S be a schema of a syntactically correct 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 ($po_S \rightrightarrows pr_S$) within schema S, if pr_S and po_S are not empty.

□

In Z, there are several operators that can be used to combine operation schemata. There are disjunction ($\vee$), implication ($\Rightarrow$), conjunction ($\wedge$), bi-implication ($\Leftrightarrow$), projection ($\upharpoonright$), sequential composition ($\fatsemi$) and piping ($\gg$). Additionally, a schema can be negated by using the NOT-operation ($\neg$). Depending on the operation the approximation to the calculation of pre- and post-conditions (see Chap. 4.4.1)differs.

The calculation is discussed in the sequel of this section. It strictly sticks to the following methodology:

1. Pre- and post-conditions of the schema are calculated on the syntactical level.
2. These pre- and post-conditions are used to identify pre- and post-condition primes.
3. These primes are used to provide rules for the identification of dependencies.

The following equation will be used several times to simplify expressions. It says that the existential quantification distributes through disjunction:

$$\exists A \bullet P \vee Q \qquad \Leftrightarrow \qquad \exists A \bullet P \ \vee \ \exists A \bullet Q \tag{4.7}$$

Let S be an operation schema of a syntactically correct Z specification. Furthermore, let pr_S be the set of predicates that are pre-condition primes and let po_S be the set of predicates that are post-condition primes. Equation (4.8) presents the schema in its horizontal form. If there is no danger of mixing identifiers, the schema can also be written in a more space saving manner (thus, the identifiers d_S and d'_S can be omitted in the predicate part):

$$S \;==\; [d_S;\; d'_S \mid pr_S(d_S);\; po_S(d_S, d'_S)] \;==\; [d_S;\; d'_S \mid pr_S;\; po_S] \tag{4.8}$$

According to Woodcock [WD96], every schema can be written as the conjunction of its pre- and post-conditions. Thus, a schema S can be written down as follows:

$$S \;==\; pre\ S \;\wedge\; post\ S \;==\; [d_S;\; d'_S \mid pr_S;\; po_S] \tag{4.9}$$

In this case pr_S is the true pre-condition of S and as po_S is the true post-condition of S it holds:

$$pr_S \;==\; \Pi\ (pre\ S) \qquad \wedge \qquad po_S \;==\; \Pi\ (post\ S)$$

As pr_S and po_S are the pre- and post-condition primes of S, the post-condition predicates are true when calculating the pre-condition of S. This means that an after-state exists. It also means that the pre-condition predicates are true when calculating the post-condition:

$$(pre\ S \;==\; [d_S \mid pr_S]) \qquad \Leftrightarrow \qquad ([d_S \mid \exists\, d'_S \bullet po_S] \;==\; [d_S \mid true]) \tag{4.10}$$

$$(post\ S \;==\; [d_S;\; d'_S \mid po_S]) \qquad \Leftrightarrow \qquad ([d_S \mid pr_S] \;==\; [d_S \mid true]) \tag{4.11}$$

The first operator to be examined is the logical NOT ($\neg$). In order to determine the sets of pre- and post-condition primes, first the pre- and post-condition predicates of a negated schema S are calculated.

$$\begin{aligned}
pre\ (\neg\, S) \;&==\; pre\ ([d_S;\; d'_S \mid \neg\, (pr_S;\; po_S)]) \\
&==\; pre\ ([d_S;\; d'_S \mid (\neg\, pr_S) \vee (\neg\, po_S)]) \\
&==\; [d_S \mid \exists\, d'_S \bullet (\neg\, pr_S) \vee (\neg\, po_S)] \\
&==\; [d_S \mid (\exists\, d'_S \bullet (\neg\, pr_S)) \vee (\exists\, d'_S \bullet (\neg\, po_S))] && \%\ (4.7) \\
&==\; [d_S \mid \neg\, pr_S \vee \exists\, d'_S \bullet \neg\, po_S] && \%\ d_s\ not\ bound
\end{aligned}$$

The pre-condition of schema $\neg\, S$ consists of the predicates in pr_S and a remaining term. Neglecting the remaining term leads to a weaker pre-condition of the schema[8].

$$pre(\neg\, S) \;\sqsubseteq\; pre_w(\neg\, S) \;== [d_S \mid \neg\, pr_S] \tag{4.12}$$

[8] The symbol $\sqsubseteq$ is borrowed from refinement calculus [WD96, p.298]. An expression $P \sqsubseteq Q$ is pronounced "P is refined by Q". The function pre_w is just a shorthand and denotes the calculation of the weakened pre-condition (by neglecting the remaining term).

The post-condition of schema $\neg\ S$ is calculated in a similar manner:

$$\begin{aligned}
post\ (\neg\ S)\ &==\ post\ ([d_S;\ d'_S \mid \neg\ (pr_S;\ po_S)]) \\
&==\ post\ ([d_S;\ d'_S \mid (\neg\ pr_S) \lor (\neg\ po_S)]) \\
&==\ [d'_S \mid \exists\ d_S \bullet (\neg\ pr_S) \lor (\neg\ po_S)] \\
&==\ [d'_S \mid (\exists\ d'_S \bullet (\neg\ pr_S)) \lor (\exists\ d'_S \bullet (\neg\ po_S))] && \%\ (4.7) \\
&==\ [d'_S \mid false \lor (\exists\ d'_S \bullet (\neg\ po_S))] && \%\ (4.11) \\
&==\ [d'_S \mid \exists\ d'_S \bullet (\neg\ po_S)]
\end{aligned}$$

The post-condition consists of a term that is solely dependent on the post-condition predicates. The post-condition can be strengthened[9]:

$$post(\neg\ S)\ \sqsubseteq\ post_s(\neg\ S)\ ==[d'_S \mid \neg\ po_S] \tag{4.13}$$

According to Def. 4.13, predicates representing before-state invariants or pre-condition predicates are taken as pre-condition primes P_{pre}. The same applies for the identification of post-condition primes: after-state invariants are taken as post-condition primes P_{post}. This leads to the following set of pre- and post-condition primes of schema $\neg\ S$:

$$\begin{aligned}
P_{pre}\ &==\ \Pi\ (pre_w\ \neg\ S)\ ==\ pr_S \\
P_{post}\ &==\ \Pi\ (post_s\ \neg\ S)\ ==\ po_S
\end{aligned}$$

As $pr_S = P_{pre}$ is the set of pre-conditions of $\neg\ S$, and as $po_S = P_{post}$ is the set of post-conditions of $\neg\ S$, the following rule for the identification of control dependencies in negated Z schemata can be provided:

Rule 4.2: Control dependencies in negated Z schemata. Let S be a schema of a syntactically correct specification, let pr_S be the set of pre-condition primes in S and let po_S be the set of post-condition primes in S. Furthermore, let $\hat{S}$ be the negated Z schema of S ($\hat{S} == \neg\ S$).

If pr_S and po_S are not empty, then there is control dependency between primes $po_{\hat{S}}$ and $pr_{\hat{S}}$ in $\hat{S}$. It holds:

$$po_{\hat{S}} \rightrightarrows_c pr_{\hat{S}}$$

□

Negated Z-schemata can be handled based on rule 4.2. Next, schema operations that combine two schemata are of interest. Let S and T be two operation schemata of a Z specification. Additionally, let pr_S be the pre-condition of schema S and

[9] The function $post_s$ is just a shorthand and denotes strengthening the post-condition by agreeing to do more than was originally required.

po_S the post-condition of S. Similarly let pr_T be the pre-condition of T and po_T the post-condition of T. As in equations (4.10) and (4.11) this implies that the post-condition exists when calculating the pre-condition of S or T. It also implies that the pre-condition evaluates to true when calculating the post-condition.

When calculating the pre-condition of a schema, disjunction is the simplest case. The pre-conditions of the schema disjunction $S \lor T$ is calculated as follows:

$$\begin{aligned}
pre\ (S \lor T) &== pre\ ([d_S;\ d'_S \mid pr_S;\ po_S] \lor [d_T;\ d'_T \mid pr_T;\ po_T]) \\
&== pre\ ([d_S;\ d'_S;\ d_T;\ d'_T \mid (pr_S;\ po_S) \lor (pr_T;\ po_T)]) \\
&== [d_S;\ d_T \mid \exists\, d'_S;\ d'_T \bullet ((pr_S;\ po_S) \lor (pr_T;\ po_T))] \\
&== [d_S;\ d_T \mid (\exists\, d'_S \bullet (pr_S;\ po_S)) \lor (\exists\, d'_T \bullet (pr_T;\ po_T))] \qquad \% \ (4.7) \\
&== [d_S;\ d_T \mid (pr_S;\ \exists\, d'_S \bullet po_S) \lor (pr_T;\ \exists\, d'_T \bullet po_T)] \\
&== [d_S;\ d_T \mid (pr_S;\ true) \lor (pr_T;\ true)] \qquad \% \ (4.9) \\
&== [d_S;\ d_T \mid pr_S \lor pr_T] \\
&== pre\ S\ \lor\ pre\ T
\end{aligned}$$

This means that pr_S and pr_T are the sole pre-conditions primes of the schema $S \lor T$. Post-conditions are calculated in a similar way:

$$\begin{aligned}
post\ (S \lor T) &== post\ ([d_S;\ d'_S \mid pr_S;\ po_S] \lor [d_T;\ d'_T \mid pr_T;\ po_T]) \\
&== post\ ([d_S;\ d'_S;\ d_T;\ d'_T \mid (pr_S;\ po_S) \lor (pr_T;\ po_T)]) \\
&== post\ ([d_S;\ d'_S;\ d_T;\ d'_T \mid ((pr_S;\ po_S) \lor pr_T);\ ((pr_S;\ po_S) \lor po_T)]) \\
&== post\ ([d_S;\ d'_S;\ d_T;\ d'_T \mid (pr_S \lor pr_T);\ (po_S \lor pr_T); \\
&\qquad (pr_S \lor po_T);\ (po_S \lor po_T)]) \\
&== [d_S;\ d'_S;\ d_T;\ d'_T \mid true;\ true;\ true;\ (po_S \lor po_T)] \qquad \% \ (4.11) \\
&== [d_S;\ d'_S;\ d_T;\ d'_T \mid po_S \lor po_T] \\
&== post\ S\ \lor\ post\ T
\end{aligned}$$

This means that po_S and po_T are the sole post-conditions of the schema $S \lor T$. In this sense, the schema can be written as

$$\begin{aligned}
S\ \lor\ T &== pre(S \lor T) \land post(S \lor T) \\
&== (pre(S) \lor pre(T)) \land (post(S) \lor post(T))
\end{aligned}$$

With that background, pre- and post-condition primes of the composed schema can be identified. As post-conditions and pre-conditions are in the same scope, post-condition primes po_S and po_T are control-dependent on both pr_S and pr_T. For the identification of dependencies between primes that are element of a Z schema disjunction this leads to the following rule:

Rule 4.3: Control dependencies in Z schema disjunctions. Let S and T be two operation schemata of a syntactically correct specification and $(S \lor T)$ a

schema disjunction. Furthermore, let pr_S be the set of pre-condition primes of S and po_S the set of post-condition primes of S. pr_T is the set of pre-condition primes of T, and po_T is the set of post-condition primes of T.

i. There is control dependency ($po_S \rightrightarrows_c pr_S$) within schema S if pr_S and po_S are not empty. There is control dependency ($po_T \rightrightarrows_c pr_T$) within schema T if pr_T and po_T are not empty.

ii. For schema $(S \vee T)$ holds that, according to Def. 4.16, there is control dependency between primes in *post* $(S \vee T)$ and primes in *pre* $(S \vee T)$.

There is control dependency between schema S and T in the case that at least $(pr_S \neq \varnothing \wedge po_T \neq \varnothing)$ or $(pr_T \neq \varnothing \wedge po_S \neq \varnothing)$. Then primes in po_T are control dependent on primes in pr_S, and primes in po_S are control dependent on primes in pr_T. This means:

$$\Pi post\ (S \vee T)\ \rightrightarrows_c\ \Pi pre(S \vee T)\ ==\ (po_S \cup po_T)\ \rightrightarrows_c\ (pr_S \cup pr_T)$$

□

In fact, post-condition primes are not strictly dependent on pr_S and pr_T, but, as both pre-conditions *potentially* decide whether the post-conditions are applied or not, there is control dependency as defined in Def. 4.11.

The next basic operation is that of a conjunction. The pre-condition of schema conjunction $S \wedge T$ is calculated as follows:

$$\begin{aligned}
pre\ (S \wedge T)\ &==\ pre\ ([d_S;\ d'_S \mid pr_S;\ po_S] \wedge [d_T;\ d'_T \mid pr_T;\ po_T]) \\
&==\ pre\ ([d_S;\ d'_S;\ d_T;\ d'_T \mid (pr_S;\ po_S;\ pr_T;\ po_T)]) \\
&==\ [d_S;\ d_T \mid \exists\, d'_S;\ d'_T \bullet pr_S;\ po_S;\ pr_T;\ po_T] \\
&==\ [d_S;\ d_T \mid (pr_S;\ pr_T) \wedge \exists\, d'_S;\ d'_T \bullet po_S;\ po_T]\ \ \%\ d_S, d_T\ not\ bound \\
&==\ pre\ S\ \wedge\ pre\ T\ \wedge\ t(po_S, po_T)
\end{aligned}$$

That means that pr_S and pr_T are part of the pre-condition primes of the schema $(S \wedge T)$, but there is also a remaining term t $(t = [d_S;\ d_T \mid \exists\, d'_S;\ d'_T \bullet po_S;\ po_T])$.

It is possible to eliminate the remaining term t if the sets d'_S and d'_T are disjunct and po_S and po_T do not share common identifiers. In that case the existential quantifier can be split into two expressions $\exists\, d'_S \bullet po_S$ and $\exists\, d'_T \bullet po_T$, and these expressions can be reduced to *true* according to equation (4.11).

In any other case it is not that simple (if even possible) to reduce t. However, the previous section introduced the idea of a syntactical approximation already. The condition is weakened by neglecting this remaining term t:

$$pre(S \wedge T) \sqsubseteq pre_w(S \wedge T) == pre\ S \wedge\ pre\ T \tag{4.14}$$

The post-condition of $S \wedge T$ can be calculated in a similar way.

$$\begin{array}{lll} post\ (S \wedge T) & == & post\ ([d_S;\ d'_S \mid pr_S;\ po_S] \wedge [d_T;\ d'_T \mid pr_T;\ po_T]) \\ & == & post\ ([d_S;\ d'_S;\ d_T;\ d'_T \mid pr_S;\ po_S;\ pr_T;\ po_T]) \\ & == & [d_S;\ d'_S;\ d_T;\ d'_T \mid true;\ po_S;\ true;\ po_T] \qquad \% \ (4.11) \\ & == & [d_S;\ d'_S;\ d_T;\ d'_T \mid po_S;\ po_T] \\ & == & post\ S\ \wedge\ post\ T \end{array}$$

This means that po_S and po_T are the sole post-conditions of the schema $S \wedge T$. In this sense, the schema can be written as

$$\begin{array}{l} S\ \wedge\ T\ ==\ pre(S \wedge T) \wedge post(S \wedge T) \\ \sqsubseteq \\ pre_w(S \wedge T) \wedge post(S \wedge T) \\ == \\ (pre(S) \wedge pre(T)) \wedge (post(S) \wedge post(T)) \end{array}$$

Again, pre-conditions and post-conditions are in the same scope. This means that all post-conditions po_S and po_T are control dependent on the pre-conditions pr_S and pr_T. This leads to the rule for the identification of control dependencies in Z schema conjunctions:

Rule 4.4: Control dependencies in Z schema conjunctions. Let S and T be two operation schemata of a syntactically correct specification. Let $(S \wedge T)$ be a schema conjunction. Furthermore, let pr_S be the set of pre-condition primes of S and po_S the set of post-condition primes of S. pr_T is the set of pre-condition primes of T and po_T is the set of post-condition primes of T.

For schema $(S \wedge T)$ holds that (according to Def. 4.16) there is control dependency between primes in $post\ (S \wedge T)$ and primes in $pre_w\ (S \wedge T)$. It holds:

i. There is control dependency $(po_S \rightrightarrows_c pr_S)$ within schema S if pr_S and po_S are not empty. There is control dependency $(po_T \rightrightarrows_c pr_T)$ within schema T if pr_T and po_T are not empty.

ii. There is control dependency between schema S and schema T in $(S \wedge T)$ if $(pr_S \neq \varnothing \wedge po_T \neq \varnothing)$ or $(pr_T \neq \varnothing \wedge po_S \neq \varnothing)$. It holds:

$$\Pi post\ (S \wedge T)\ \rightrightarrows_c\ \Pi pre_w(S \wedge T)\ ==\ (po_S \cup po_T)\ \rightrightarrows_c\ (pr_S \cup pr_T)$$

□

With the identification of pre- and post-conditions for conjunction and disjunction, most of the schema operations defined in Z can be dealt with. An implication can be re-written as a disjunction which leads to the following set of pre-condition primes pr and post-condition primes po:

$$\begin{aligned}
&S \;\Rightarrow\; T \;==\; (\neg\, S \;\lor\; T)\\
&pr \;==\; \Pi\,(pre\;(S \;\Rightarrow\; T)) \;==\; \Pi\,(pre(\neg\, S \;\lor\; T))\\
&\quad ==\; \Pi\,(pre_w(\neg\, S) \lor pre(T)) \;==\; pr_S \cup pr_T\\
&po \;==\; \Pi\,(post\;(S \;\Rightarrow\; T)) \;==\; \Pi\,(post(\neg\, S \;\lor\; T))\\
&\quad ==\; \Pi\,(post_s(\neg\, S) \lor post(T)) \;==\; po_S \cup po_T
\end{aligned}$$

Bi-implication can be re-written as a combination of a conjunction and two disjunctions. Pre-condition primes pr and post-condition primes po are calculated as follows:

$$\begin{aligned}
&S \;\Leftrightarrow\; T \;==\; (\neg\, S \;\lor\; T) \land (S \;\lor\; \neg\, T)\\
&pr \;==\; \Pi(pre\;(S \;\Leftrightarrow\; T)) \;==\; \Pi(pre((\neg\, S \;\lor\; T) \land (S \;\lor\; \neg\, T)))\\
&\quad ==\; \Pi(pre(\neg\, S \;\lor\; T) \land pre(S \;\lor\; \neg\, T)) \;==\; pr_S \cup pr_T\\
&po \;==\; \Pi(post\;(S \;\Leftrightarrow\; T)) \;==\; \Pi((\neg\, S \;\lor\; T) \land (S \;\lor\; \neg\, T))\\
&\quad ==\; \Pi(post(\neg\, S \;\lor\; T) \land post(S \;\lor\; \neg\, T)) \;==\; po_S \cup po_T
\end{aligned}$$

Schema projection is more complex to dissolve. In Z a schema projection $S \upharpoonright T$ can be re-written as a schema conjunction $S \land T$ where those identifiers $x_1, .., x_n$ of S are hidden which do not appear in T:

$$S \;\upharpoonright\; T \;==\; (S \;\land\; T) \setminus (x_1, .., x_n)$$

In fact two situations can occur when combining two schemata via schema projection:

1. If all identifiers of S are also declared in T, then the projection equals the schema conjunction. In this case the set of pre-condition primes is $(pr_S \cup pr_T)$. The set of post-condition primes is $(po_S \cup po_T)$.

2. If there are identifiers in S that are not declared in T, then some (or even all) identifiers of S are hidden. In that case some or all predicates in S are hidden by using an existential quantifier. To ease the calculation of pre- and post-conditions the declaration sections of S and T are split into several sets: $\hat{d_S} = d_S \setminus d_T$, $d = d_S \cap d_T$, $\hat{d_T} = d_T \setminus d_S$, $\hat{d'_S} = d'_S \setminus d'_T$, $d' = d'_S \cap d'_T$ and $\hat{d'_T} = d'_T \setminus d'_S$. With this separation of identifiers the schemata S and T can be re-written as follows:

$$\begin{aligned}
&S \;==\; [\hat{d_S};\; d;\; \hat{d'_S};\; d' \mid pr_S(\hat{d_S}, d);\; po_S(\hat{d_S}, d, \hat{d'_S}, d')]\\
&T \;==\; [\hat{d_T};\; d;\; \hat{d'_T};\; d' \mid pr_T(\hat{d_T}, d);\; po_T(\hat{d_T}, d, \hat{d'_T}, d')]
\end{aligned}$$

The pre-conditions of a schema projection is calculated as follows:

$$\begin{aligned} pre((S \wedge T) \setminus (\hat{d_S}, \hat{d'_S})) \;==\; & [d;\; \hat{d_T} \mid \exists\, d';\; \hat{d'_T} \bullet \exists\, \hat{d_S};\; \hat{d'_S} \bullet \\ & pr_S(\hat{d_S}, d);\; po_S(\hat{d_S}, d, \hat{d'_S}, d');\\ & pr_T(d, \hat{d_T});\; po_T(d, \hat{d_T}, d', \hat{d'_T})] \\ == [d;\; \hat{d_T} \mid & pr_T(d, \hat{d_T}) \wedge \exists\, \hat{d_S} \bullet pr_S(\hat{d_S}, d)] \end{aligned}$$

If the pre-condition is weakened, then the set of pre-condition primes is pr_T. Nevertheless, the pre-condition of S has to exist in order to satisfy the pre-condition of $S \upharpoonright T$. Thus, the set of relevant primes is $pr_S \cup pr_T$.

The post-conditions of a schema projection is calculated as follows:

$$\begin{aligned} post((S \wedge T) \setminus (\hat{d_S}, \hat{d'_S})) \;==\; & [d';\; \hat{d'_T} \mid \exists\, d;\; \hat{d_T} \bullet \exists\, \hat{d_S};\; \hat{d'_S} \bullet \\ & pr_S(\hat{d_S}, d);\; po_S(\hat{d_S}, d, \hat{d'_S}, d');\\ & pr_T(d, \hat{d_T});\; po_T(d, \hat{d_T}, d', \hat{d'_T})] \\ == [d';\; \hat{d'_T} \mid & \exists\, d;\; \hat{d_T};\; \hat{d_S};\; \hat{d'_S} \bullet \\ & po_S(\hat{d_S}, d, \hat{d'_S}, d');\; po_T(d, \hat{d_T}, d', \hat{d'_T})] \end{aligned}$$

If the post-condition is strengthened, then the set of post-condition primes is $(po_S \cup po_T)$.

In both cases the post-condition contains the post-condition predicates of S and T. It seems to be natural to take the set $po_S \cup po_T$ as the set of primes which are post-condition primes of schema $S \upharpoonright T$.

In the case of the pre-condition, pr_T definitely contributes to the set of pre-condition primes. pr_S also has to be part of the pre-condition. Thus, $pr_S \cup pr_T$ is the set of primes which contains the pre-condition primes of schema $S \upharpoonright T$.

Analogous to Rule 4.3, control dependencies in schema implications, bi-implications and projections are identified as follows:

Rule 4.5: Control dependencies in Z schema (bi-)implications and projection. Let S and T be two operation schemata of a syntactically correct specification and $(S \diamondsuit T)$ a schema operation where $\diamondsuit$ is either a implication, a bi-implication, or a projection. Furthermore, let pr_S be the set of pre-condition primes of S and po_S the set of post-condition primes of S. pr_T is the set of pre-condition primes of T and po_T is the set of post-condition primes of T. It holds:

i. There is control dependency $(po_S \rightrightarrows_c pr_S)$ within schema S if pr_S and po_S are not empty. There is control dependency $(po_T \rightrightarrows_c pr_T)$ within schema T if pr_T and po_T are not empty.

ii. There is control dependency between schema S and schema T, if at least $(pr_S \neq \varnothing \wedge po_T \neq \varnothing)$ or $(pr_T \neq \varnothing \wedge po_S \neq \varnothing)$. It holds:

$$\begin{array}{l} \Pi(post\ (S \diamondsuit T)) \ \rightrightarrows_c \ \Pi(pre(S \diamondsuit T)) \\ == \\ (po_S \cup po_T) \ \rightrightarrows_c \ (pr_S \cup pr_T) \end{array}$$

□

Five schema operations $(\wedge, lor, \Rightarrow, \Leftrightarrow, \restriction)$ have been dealt with. Remain the two schema operations of sequential composition ($\fatsemi$) and schema piping ($\gg$). Composition is defined as follows:

$$S \fatsemi T == (S[d^+/d'] \wedge T[d^+/d]) \setminus (d^+) \tag{4.15}$$

After state identifiers d' of the first schema and before state identifiers of the second schema are renamed to d^+ (thus, the two schemata are related by an intermediate state). These identifiers are finally hidden by using existential quantification.

Renaming and identifier hiding also happen when schema piping is applied:

$$S \gg T == (S[d^+/d!] \wedge T[d^+/d?]) \setminus (d^+) \tag{4.16}$$

In this case identifiers $d!$, representing the output of S, and identifiers $d?$, representing the input of T, are renamed to d^+. Again, these identifiers are hidden by using an existential quantification.

For a meaningful composition or piping the related sets of identifiers have to match. This means that the intersection of the set of identifiers in d'_S and d_T has to be non-empty: $\hat{d} = d'_S \cap d_T$.

In the case of a composition of S and T, $\hat{d}$ contains after-state identifiers of S that are also before-state identifiers of T. In the case of schema piping, $\hat{d}$ contains output identifiers of S that are also input identifiers of T.

For the following calculation of pre- and post-conditions, the declaration sections of S and T are split into three parts. Due to the state change, parts of the identifiers have to be renamed. For this reason pre- and post-condition predicates contain the full list of identifiers they depend upon. For schemata S and T holds:

$$S == [d_S;\ \hat{d};\ d'_S \mid pr_S(d_S);\ po_S(d_S, \hat{d}, d'_S)] \tag{4.17}$$

$$T == [d_T;\ \hat{d};\ d'_T \mid pr_T(d_T, \hat{d});\ po_S(d_T, \hat{d}, d'_T)] \tag{4.18}$$

Thus, the previously introduced sets d'_S and d_T are split into two sets ($\hat{d}$,d'_S) and (d_T, $\hat{d}$). It holds that $\hat{d} \cap d'_S = \varnothing$ and $d_T \cap \hat{d} = \varnothing$. However, pr_S and po_S still denote the pre- and post-conditions of schema S and pr_T and po_T denote the pre- and post-conditions of schema T. This implies:

$$pre\ S\ ==\ [d_S \mid pr_S(d_S)] \quad \Leftrightarrow \quad [d_S \mid \exists\, \hat{d};\ d'_S \bullet po_S(d_S, \hat{d}, d'_S)]\ ==\ true \tag{4.19}$$

$$post\ S\ ==\ [d_S;\ \hat{d};\ d'_S \mid po_S(d_S, \hat{d}, d'_S)] \quad \Leftrightarrow \quad [d_S \mid pr_S(d_S)]\ ==\ true \tag{4.20}$$

$$pre\ T\ ==\ [d_T;\ \hat{d} \mid pr_T(d_T, \hat{d})] \ \Leftrightarrow\ [d_T;\ \hat{d} \mid \exists\, d'_T \bullet po_T(d_T, \hat{d}, d'_T)]\ ==\ true \tag{4.21}$$

$$post\ T\ ==\ [d_T;\ \hat{d};\ d'_T \mid po_T(d_T, \hat{d}, d'_T)] \ \Leftrightarrow\ [d_T;\ \hat{d} \mid pr_T(d_T, \hat{d})]\ ==\ true \tag{4.22}$$

With that, pre- and post-conditions of schemata $S \fatsemi T$ and $S \gg T$ can be calculated. The operator $\heartsuit$ is used to denote either schema composition or schema piping:

$$\begin{aligned}
pre\ (S \heartsuit T)\ ==&\ pre\ ([d_S;\ \hat{d};\ d'_S \mid pr_S(d_S);\ po_S(d_S, \hat{d}, d'_S)] \\
&\quad \heartsuit\ [d_T;\ \hat{d};\ d'_T \mid pr_T(d_T, \hat{d});\ po_T(d_T, \hat{d}, d'_T)]) && \%\ (1)\\
==&\ pre\ [d_S;\ d'_S;\ d_T;\ d'_T \mid \exists\, \hat{d}^+ \bullet \\
&\quad pr_S(d_S);\ po_S(d_S, \hat{d}^+, d'_S); \\
&\quad pr_T(d_T, \hat{d}^+);\ po_T(d_T, \hat{d}^+, d'_T)] && \%\ (2)\\
==&\ [d_S;\ d_T \mid \exists\, d'_S;\ d'_T \bullet \exists\, \hat{d}^+ \bullet && \%\ (3)\\
&\quad pr_S(d_S);\ po_S(d_S, \hat{d}^+, d'_S); \\
&\quad pr_T(d_T, \hat{d}^+);\ po_T(d_T, \hat{d}^+, d'_T)] \\
==&\ [d_S \mid pr_S(d_S) \wedge \exists\, \hat{d}^+;\ d'_T;\ d'_S \bullet && \%\ (4)\\
&\quad po_S(d_S, \hat{d}^+, d'_S);\ pr_T(d_T, \hat{d}^+);\ po_T(d_T, \hat{d}^+, d'_T)] \\
==&\ pre\ S\ \wedge\ t(po_S, pr_T, po_T)
\end{aligned}$$

The calculation is straight forward: (1) First the operator is dissolved by a conjunction of S and T, and (2) hides the inter-state identifiers. (3) Then the pre-condition calculation is made explicit by hiding the after-state identifiers. The predicates are re-arranged. The pre-condition of S is extracted from the quantifiers as there is no bound identifier in pr_S. (4) This leads to a pre-condition containing the pre-condition predicate of S and a remaining term t. In any case, the remaining term t depends on inter- and after state identifiers. As we are only interested in pre-state predicates, the remaining term is neglected again. The pre-condition is weakened.

$$
\begin{array}{l}
pre\ (S \heartsuit T)\ ==\ pre\ S\ \wedge\ t(po_S, pr_T, po_T) \\
\sqsubseteq \\
pre_w\ (S \heartsuit T)\ ==\ pre(S)\ ==\ [d_S \mid pr_S(d_S)]
\end{array}
$$

Post-conditions are calculated in the same way. Again, an inter-state is built by first renaming identifiers, combining the schemata and then hiding inter-state identifiers:

$$
\begin{array}{lll}
post\ (S \heartsuit T) & == & post\ ([d_S;\ \hat{d};\ d'_S \mid pr_S(d_S);\ po_S(d_S, \hat{d}, d'_S)] \\
& & \quad \heartsuit\ [d_T;\ \hat{d};\ d'_T \mid pr_T(d_T, \hat{d});\ po_S(d_T, \hat{d}, d'_T)]) \\
& == & post\ [d_S;\ d'_S;\ d_T;\ d'_T \mid \exists\, \hat{d}^+ \bullet \\
& & \quad pr_S(d_S);\ po_S(d_S, \hat{d}^+, d'_S); \\
& & \quad pr_T(d_T, \hat{d}^+);\ po_T(d_T, \hat{d}^+, d'_T)] \\
& == & post\ [d_S;\ d'_S, d_T;\ d'_T \mid pr_S(d_S) \wedge \exists\, \hat{d}^+ \bullet \\
& & \quad po_S(d_S, \hat{d}^+, d'_S); \\
& & \quad pr_T(d_T, \hat{d}^+);\ po_T(d_T, \hat{d}^+, d'_T)] \\
& == & [d'_S;\ d'_T \mid \exists\, d_S;\ d_T \bullet \exists\, \hat{d}^+ \bullet \qquad \% \ (4.20) \\
& & \quad po_S(d_S, \hat{d}^+, d'_S); \\
& & \quad pr_T(d_T, \hat{d}^+);\ po_T(d_T, \hat{d}^+, d'_T)] \\
& == & [d'_S;\ d'_T \mid \exists\, d_S;\ d_T;\ \hat{d}^+ \bullet \qquad \% \ (4.22) \\
& & \quad po_S(d_S, \hat{d}^+, d'_S);\ po_T(d_T, \hat{d}^+, d'_T)] \\
& == & t(po_S, po_T)
\end{array}
$$

In this case the post-condition is a combination of the post-condition of S and the post-condition of T. The remainder cannot be simplified without semantic analysis, and it can also not be split into a combination of separate post-conditions of S and T (as they share common identifiers $\hat{d}^+$). On the other hand it is possible to refine the post-condition by strengthening the condition.

$$
\begin{array}{l}
post\ (S \heartsuit T)\ == [d'_S;\ d'_T \mid \exists\, d_S;\ d_T;\ \hat{d}^+ \bullet \\
\qquad\qquad po_S(d_S, \hat{d}^+, d'_S);\ po_T(d_T, \hat{d}^+, d'_T)] \\
\sqsubseteq\ (post - condition\ strengthening) \\
post_s\ (S \heartsuit T)\ == [d_S;\ d'_S;\ \hat{d}^+;\ d_T;\ d'_T \mid \\
\qquad\qquad po_S(d_S, \hat{d}^+, d'_S);\ po_T(d_T, \hat{d}^+, d'_T)] \\
== \\
post\ (S)\ \wedge\ post\ (T)
\end{array}
$$

In this case, the strengthened post-condition consists of the post-condition of S and the post-condition of T. Control dependency is defined between post- and pre-condition primes in the schema. The pre-condition prime of $S \heartsuit T$ is pr_S, the post-condition primes of $S \heartsuit T$ are po_S and po_T. As po_S is element of schema S, there

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

Tab. 4.1: Control dependency calculation differs, depending on the type of schema operation. This table provides an overview of relevant primes and their related pre- and post-condition considerations.

is control dependency between po_S and pr_S. Additionally po_T is control dependent on pr_S. This leads to the rule for the identification of control dependencies within schema composition and piping:

Rule 4.6: Control dependencies in schema composition and piping. Let S and T be two operation schemata of a syntactical correct specification and $(S \heartsuit T)$ a schema operation where $\heartsuit$ is either a composition or a piping operation. Furthermore, let pr_S be the set of pre-condition primes of S and po_S the set of post-condition primes of S. pr_T is the set of pre-condition primes of T and po_T is the set of post-condition primes of T. It holds:

i. There is control dependency $(po_S \rightrightarrows_c pr_S)$ within schema S if pr_S and po_S are not empty. There is control dependency $(po_T \rightrightarrows_c pr_T)$ within schema T if pr_T and po_T are not empty.

ii. There is control dependency between schema S and schema T if $pr_S \neq \varnothing \wedge po_T \neq \varnothing$. It holds:

$$\Pi(post_s\ (S \heartsuit T)) \rightrightarrows_c \Pi(pre(S \heartsuit T)) == (po_S \cup po_T) \rightrightarrows_c pr_S$$

□

With the above rules for the identification of control dependencies it is possible to deal with all types of Z schema operations. However, only in one case ($\vee$) can the calculation of pre- and post-conditions of the compound schemata be traced back to the (simpler) calculation of pre- and post-conditions of the involved schemata.

In all other cases semantic errors are made. In five cases ($\neg\ , \wedge, \Rightarrow, \Leftrightarrow, \upharpoonright$) the pre-condition is weakened. This means that a little bit more cases are handled by the schema than originally intended. In five cases ($\neg\ , \Rightarrow, \Leftrightarrow, \fatsemi, \gg$) the post-condition is strengthened. This means that the schema is agreeing to do more than originally intended. Table 4.1 summarizes the involved operations, as well as the pre- and post-conditions that are taken as a basis for the calculation of involved primes.

Fig. 4.4 (at the beginning of Chap. 4.4.2) presents three examples for the identification of control dependencies in specifications. Within schema-box $S1$, all post-condition primes are control dependent on pre-condition primes (case (a)). Case (b) demonstrates the logical combination of two schemata $S2$ and $S3$. Here, both post-condition primes are control dependent on both pre-condition primes. The situation is different, if an explicit sequential ordering is carried out (case (c)). Then the post-condition primes of the second schema are control dependent on both pre-condition primes, but the pre-condition prime of the second schema has no influence on the post-condition of the first schema.

However, the above syntactic approximation is quite accurate. Concerning the weakened pre-conditions, the neglected term t consists of post-condition predicates solely (ensuring that an after-state exists). In the context of looking for possible pre-condition primes, an incessantly existing after-state is not that important. There might be cases where the pre-condition does not hold, but in any case the weakened terms have to be considered.

Concerning the strengthened post-condition, no term is neglected. In fact, the refinement is used to produce a much stronger post-condition on the semantic level. On the syntactic level the same primes are involved again. In the true post-condition they are potentially involved, in the strengthened post-condition they are *definitely* involved.

With the notion of control, the definition of data dependency gets possible, too. Def. 4.9 points to the parts to look for: 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 of a declaration or at the left side of a schema expression. In the declaration "$p? : Person$", the identifier p is said to be declared. Another example is the expression "$ZeroOp == Enter \wedge Leave$", where two operation schemata ($Enter, Leave$) are combined to declare a new operation ($ZeroOp$). The declaration of an identifier can be compared to type definitions within programs.

- *defined*, if the identifier is decorated and appears at the left side of a value assignment. In the expression "$p! = newperson$", the identifer p is said to be

defined. The declaration of an identifier can be compared to value assignments within programs.

- *used*, if it is neither declared nor defined. In the above examples, the identifers *Person*, *Enter*, *Leave* and *newperson* are used. It is important to distinguish between decorated and undecorated identifiers. In the expression $p = \{\}$ the identifier p is used. There is no value assignment and no definition.

In Z, definitions and uses of data elements are easy to detect. If a specification prime contains an equation or assignment and a left-hand side identifier (a literal denoting a data element) which is decorated by a $\boxed{'}$ or $\boxed{!}$, it is said to have a value assignment (it is defined). If it is not decorated or decorated with a question-mark $\boxed{?}$, it is a reference to a data element (it is used). Based on this terminology, data-dependency in Z specifications (see Def. 4.19) is defined as follows:

Definition 4.19: 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. However, it is just a means to an end. Dependencies alone to not suffice to sustain the comprehension process. Based on dependencies, specification abstractions can be calculated, but it is not only abstraction that counts. The subsequent section introduces the idea of visualizing dependencies properly and thus leads on to the topic of the next chapter, the augmented specification relationship net.

4.5 The Need for an Alternative Representation

Specification visualization class SV1 (Chap. 4.1.1) comprises tools for writing, reading and browsing specifications. The first subclass (SV1-a) can now be supported by abstractions based on the above definitions of dependencies. However, what is still missing is support for proper visualization (identified as specification class SV1-b).

The concepts of specification primes, slices and chunks support comprehension tasks rather in the sense of focusing and locality. On the other side, specification visualization class SV1-b (see Chap. 4.1.1) demands support for alternative, more graphical views onto the specification. Here, the idea is

- to show relationships (both syntactical and structural) hidden behind the specification and
- to focus on specific properties of a specification.

What is a graphical view? A Colored Petri Net [Jen97] is such an example of a formal specification. The net is defined by using statements based on mathematics. It is common to check the specification syntactically, to proof properties of the net and even to animate [LW98, KMW98] the net[10]. Even if there is a textual representation of the net (which is the basis for proofs), the most widely used representation is its graphical form. It focuses on structural relationships and abstracts from the nitty-gritty details of variables by using (colored) tokens. The advantage is that using the graphical form of the net's representation, its (sub-) structures, similarities between its (sub-) structures and special properties can easily be deduced.

The same considerations hold for other types of specification languages. In Z or in VDM, there are relationships between parts of the specification. There are structures and substructures, each of which interrelated by syntactic and semantic dependencies. For a proper visualization it is necessary to make them explicit. Whereas Chap. 4.1 elaborated on setting the focus on specific parts of the specification by *making use of dependencies*, the basic idea now is to *make the dependencies explicit*. The next chapter presents an alternative view to model-oriented specifications. Moreover, this type of a specification's representation also simplifies the calculation of dependencies. The alternative representation of the specification is called *Augmented Specification Relationship Net*.

4.6 Summary

The objective of this work is to make specifications more comprehensible. So this chapter starts with an overview of existing tools and approaches. It divides them into three specification visualization classes (SV1 to SV3) and explains why existing tools are not fully applicable to specification comprehension tasks. Basically there is

- missing knowledge about useful components (substructures) of a specification,
- no support for focusing on specific parts of a specification,
- a lack in the visualization of specifications and substructures of and within specifications.

[10] Design/CPN is a tool for defining and animating Colored Petri Nets. See http://www.daimi.aau.dk/designCPN for more details. Web page last visited: Nov. 2003.

The chapter then focuses on components of specifications and provides definitions for specification abstractions. Based on these definitions, the first question in the motivation section of this work is answered: abridgements (specification primes, fragments, chunks and slices) are practicable for most common specification comprehension activities. The basic mechanism is, as with most program comprehension approaches, to look for control and data-dependencies. This chapter therefore provides an approach for the identification of these types of dependencies in Z-specifications.

Control dependencies in Z are located in a schema between pre- and post-condition primes. The calculation of pre- and post-conditions is a time-consuming task. Thus, the calculation is skipped by directly taking before-state predicates as pre-conditions and after-state predicates as post-conditions. This chapter demonstrates that this approach is a useful approximation to the semantical analysis, but is also discusses the semantic error that is made.

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 again reduced to sets of before- and after-state primes. Based on these considerations, this chapter then presents rules for detecting control dependencies within a Z schema and four rules for detecting control dependencies within composed schemata.

With the presented specification abridgements the demand for partiality is being met. But partiality is not the only solution to specification comprehension. Existing relationships between components of specifications also have to be taken into consideration. Thus, this chapter closes with the observation that there is need for proper (graphical) visualization.

5. AUGMENTED SPECIFICATION RELATIONSHIP NET

Look at every path closely and deliberately.
Try it as many times as you think necessary.
Then ask yourself, and yourself alone, one question:
Does this path have a heart?
If it does, the path is good; if it doesn't, it is of no use.

Carlos Castanedas, The Teaching of Don Juan

The objective of this work is to sustain the process of specification comprehension. One solution is to reduce the complexity of the formal specifications at hand. Here, specification slices, chunks and fragments enable focusing on a specific problem. However, the identification is based on dependency analysis.

Tearing specifications into pieces is not the only solution. Another solution is to provide additional and supporting information for the problem at hand. Connections and dependencies (of different types) between parts of the specification can add valuable information. This approach makes implicit information explicitly available. However, the identification is based on dependency analysis again.

Both approaches are based on dependency analysis. As has already been mentioned in previous chapters the identification of dependencies in specifications is hard. The declarative nature and the freedom in placing specification primes impede the reconstruction of hidden control and data-dependencies. Conventional program dependency analysis cannot be applied. Nevertheless, control and data dependencies are important, and the previous chapter presented an approach for their identification.

This chapter presents an alternative view on state-based specifications: the augmented specification relationship net (ASRN for short). It can be used to detect dependencies and to visualize a specification's structure as well as dependencies. Whereas the net is generic, the necessary transformation dependents on the specification language at hand. To demonstrate this transformation, this chapter presents rules which are necessary to establish an isomorphic mapping between Z specifications and augmented specification relationship nets.

5.1 Motivation for Specification Transformation

Specification languages have built in clues to convey semantics. As with programming languages, syntax *and* layout of specification languages[1] try to sustain the process of creation. Chap. 3 explains why these characteristics are detrimental for detecting programming-language-like dependency types. So, how can hidden structural properties of written specifications be detected? And how can dependencies be identified simultaneously? A look at the state-of-the-art of other disciplines helps:

- 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 [ASU86, p.49] which enables the construction of a program dependency graph, and which ultimately facilitates program dependency analysis[2].
- For concurrent logic programming languages Zhao showed in [Zha96] that a transformation of concurrent logic programs to a graph is useful for dependency detection. He also showed that this graph can be used as a basis for metrics calculation [ZCU96].

Following these ideas, it seems appropriate to transform the specification into a graph, analyze dependencies and

(i) either use the graph to visualize relationships hidden in the specification, or

(ii) apply filter/selection operations to the graph and perform a back-transformation of the resulting sub-graph.

As we will see later in this chapter, the graph presents itself as an intermediate aid to generate specification abstractions.

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. In addition to that 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 peruser which representation s/he chooses for the problem at hand.

[1] Layout and syntactic possibilities are manifold for different approaches. There are state-based approaches focusing on the model of the system behavior, property oriented approaches focusing on pre- and post-conditions and process algebras focusing on support for concurrent, real-time and distributed systems [BH97].

[2] Program dependency analysis has two main research streams: performing optimizations and support for comprehension and maintenance approaches.

Using a syntax tree seems to be an obvious choice, but it is not the most appropriate one. Abstract syntax trees are commonly used during the analysis phase where tokens of a source program are grouped into grammatical phrases[3]. An abstract syntax tree is less appropriate for the following reasons [MB03]:

- An abstract syntax tree follows the most dominant structuring principle of programming languages: control. The notion of *control is not dominant* in declarative specification languages. However, disregarding control (to some respect) is a key principle of several specification languages, and this key property should be kept as far as possible.

- An abstract syntax tree *focuses* on the syntax of the specification and *not on the layout*. In conventional programs layout has no semantic meaning, but dismantled layout distorts readability. When creating an abstract syntax tree for a specification, the syntactic relationships between grammatical phrases are made explicit. E.g. sub-expressions as children of nodes are combined to nodes denoting expressions which are itself children of nodes denoting schema expressions, and so on. This means that the syntactic structure of the specification is dominant and not the overall layout.

- There is a certain degree of freedom in gluing specification elements together. An abstract syntax tree, on the other side, introduces hierarchies, *hierarchies* that are not predominant in specifications. Again, any strict ordering (that an abstract syntax tree suggests) would be detrimental for the overall comprehension process. In the abstract tree there is a clear ordering of the grammatical phrases. This ordering is stipulated by the syntactic rules of the language definition and the automaton that is creating the tree. In most cases the tree is created by scanning the specification text in top-down manner, creating a tree reflecting exactly the ordering of primes.

For these reasons a net is introduced [Bol02]. The structural information of the specification 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. The net will be expounded in more detail in Chap. 5.2. The SRN is extended by additional information dependent on the specification language at hand. An *eSRN* is created. It deals with layout and formatting and will be discussed in Chap. 5.3.1. The eSRN is then augmented by the use of identifiers and thus captures the explicit semantics of the specification. The resulting net is called

[3] These grammatical phrases are represented in a tree called *parse tree*. An abstract syntax tree is a compressed representation of the parse tree where each node represents an operator or non-terminal and where the children of a node represent the operands or terminals contributing to the parent node.

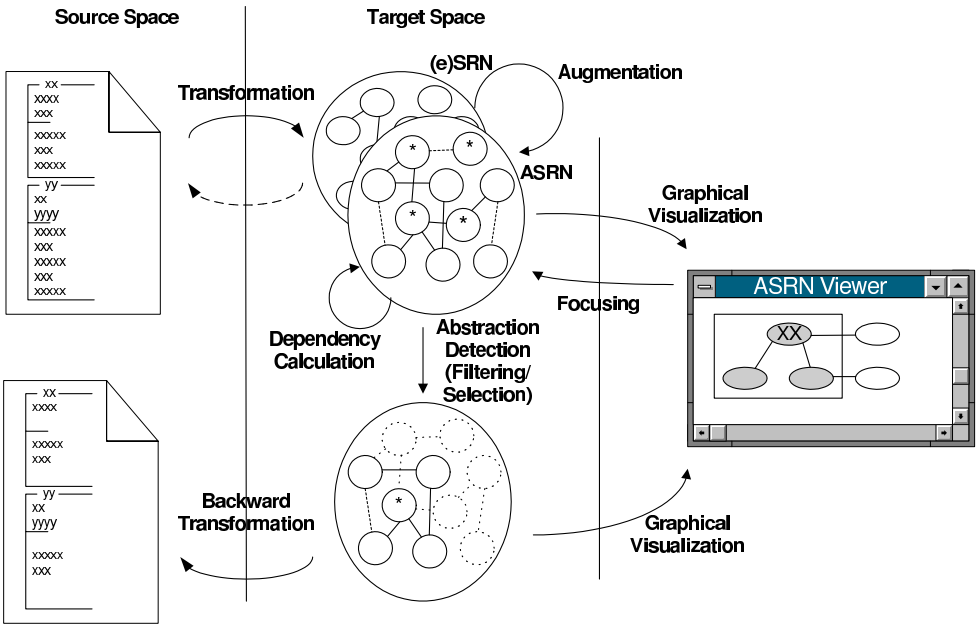


Fig. 5.1: By using a transformation operation, a specification (upper left) is ported from the source space representation (textual form) to its graphical eSRN representation. This net is augmented by definition and use information of identifiers, which leads to the ASRN. This net can be used to calculate and visualize relations hidden in the specification. By applying filter and selection functions, specification components can be identified. A backward transformation leads to the textual form of the such generated specification abstraction.

Augmented Specification Relationship Net (*ASRN*). It will be expounded in more detail in Chap. 5.4.

Fig. 5.1 demonstrates the general idea without going too much into detail for the moment. Firstly, all primes of the specification are transformed into an SRN. Secondly, language-dependent structural information is added to produce an eSRN. Then the net is augmented by declaration, definition and use information – resulting in an ASRN. On the basis of the ASRN dependencies (of different types) are calculated and, occasionally, visualized. The ASRN is used for the following four tasks:

1. *Analysis.* Based on reachability definitions, various types of relationships between vertices can be identified in the net. As will be discussed in this chapter later, programming-like dependencies can easily be calculated.

2. *Focusing.* Filter and selection functions (based on dependencies that are iden-

tified in the analysis step) can be applied. They automatically construct context around a given point of interest. As we will see in this chapter later, chunks and slices can be identified this way. As the transformation is defined in a bijective manner, a backward transformation is possible. This leads to the textual representation of the filtered specification (e.g. a specification fragment, chunk, or slice).

3. *Visualization.* Experiments showed [MB03] that the layout of the net contains a lot of relevant information concerning the relationships between parts of the specification. The ASRN can be visualized, and this view onto the net can be used (i) to ease setting the point of interest, (ii) to browse the specification, and (iii) to visualize structural properties of the specification.

4. *Metrics.* Last but not least, the transformed specification (the augmented SRN) can be used for metrics calculation [MB03]. Details are provided in the subsequent chapter.

5.2 The Specification Relationship Net

This section presents the basic formal definition of the SRN, the specification relationship net. Firstly, a common mathematical background is provided, and secondly, properties of the SRN are presented. This and the subsequent sections provide quite a lot of definitions. To ease orientation the most important terms are set in bold letters.

5.2.1 Basic Definitions

The definition of a specification relationship net is based on the definition of a bipartite graph. The general idea is that primes in a specification are represented as special vertices in the graph, and relationships between these primes are expressed via typed arcs.

Definition 5.1: Bipartite graph. A bipartite graph is an ordered pair (V, A), where V is a finite set of elements called *vertices*, and A is a finite set of elements of the Cartesian product $V \times V$, called *arcs* (A is a binary relation on V, $A \subseteq V \times V$).

A *simple bipartite graph* is a bipartite graph (V, A), such that no $(v, v) \in A$ for any $v \in V$. For any arc $a_i = (v_1, v_2) \in A$, $\alpha(a_i) = v_1$ is called the *initial vertex* of the arc, and $\omega(a_i) = v_2$ is called *terminal vertex* of the arc.

In an *SRN*, arcs can belong to different classes. Thus, the notion of an arc-classified bipartite graph is provided in the following definition. The graph consists of a set of vertices V and $n-1$ sets of arcs (thus, it makes up an n-tuple):

Definition 5.2: Arc-classified bipartite graph. An arc-classified bipartite graph is an n-tuple $(V, A_1, A_2, ..., A_{n-1})$ such that every pair (V, A_i) $(i \in \{1..(n-1)\})$ is a bipartite graph and $A_i \cap A_j = \varnothing$ for $i, j \in \{1..(n-1)\}$, and $i \neq j$.

A so-called *simple arc-classified bipartite graph* is an arc-classified bipartite graph $(V, A_1, A_2, ..., A_{n-1})$, such that no $(v, v) \in A_i$ $(i \in \{1..(n-1)\})$ for any $v \in V$.

Def. 5.2 presents the basis, upon which the specification relationship net, a bipartite graph containing arcs of special (but mutual exclusive) classes, is defined. A sequence of connected arcs in the graph is called a path:

Definition 5.3: Path. A *path* p in an arc-classified bipartite graph $(V, A_1, A_2, ..., A_{n-1})$ is a sequence of arcs $a_1, a_2, ..., a_m$ $(a_i \in \bigcup_{j=1..(n-1)} A_j$ for $i \in \{1 \,..\, m\})$, where for every i $(1 \leq i < m)$ the terminal vertex of a_i is the initial vertex of a_{i+1}.

m is called the *length* of path p. $\alpha(p) = \alpha(a_1) = v_i$ is called initial vertex of path p. $\omega(p) = \omega(a_m) = v_t$ is called terminal vertex of path p. A vertex v_k is said to be *element* of the path if it holds: v_k is the initial vertex of path p, or v_k is the terminal vertex of one of the arcs $a_1, ..., a_m$ of p. Let v_i be the initial vertex of path p and v_m the terminal vertex of p. Then the path is called a path from v_1 to v_m.

The general idea of an SRN is that primes of a specification are represented in such a way that a higher-level prime is a sub-graph with a unique syntactic start-vertex and a unique syntactic end-vertex. A prime object of the specification is assigned one-to-one to a prime vertex in the net. The SRN consists of vertices representing either primes, start-, or end-vertices. Arcs represent the logical structure of the specification. Primes can be combined via conjunctions, disjunctions, or by an explicit sequential operation. Thus, they are classified as AND-, OR-, or sequential-control arcs. In the subsequent definition, *in-degree*(v) is an operation that returns the number of arcs ending at vertex v.

Definition 5.4: Specification relationship net. A specification relationship net (*SRN* for short) of a specification is an 8-tuple $(V, V_{pr}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$ where $(V, A_c, A_{and}, A_{or})$ is a simple arc-classified bipartite graph. V is the set of all vertices in the net, A_c is the set of sequential-control arcs, A_{and} is the set of AND-control arcs, and A_{or} is the set of OR-control arcs ($A_c \subset V \times V$, $A_{and} \subseteq V \times V$, $A_{or} \subseteq V \times V$). Furthermore, it holds that:

i. $V_{pr} \subset V$ is a set of vertices called prime vertices such that $V_{pr} = \{v \mid v \textit{ represents a prime of the specification}\}$.

ii. $V_{start} \subset V$ where $V_{start} \cap V_{pr} = \varnothing$ is a set of vertices called start vertices.

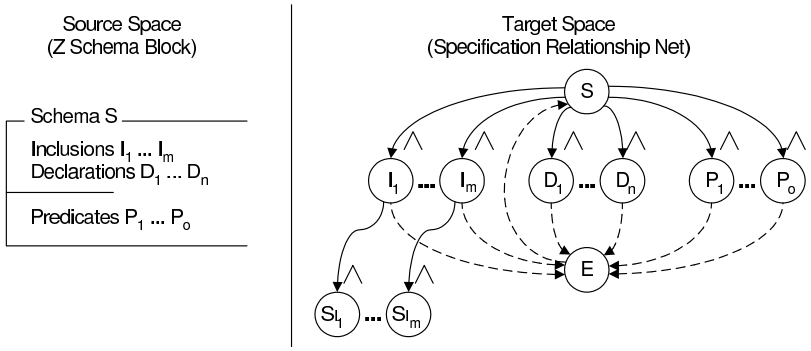


Fig. 5.2: By applying a transformation function, a Z specification (left side) is transformed from the source space representation (textual form) into its SRN representation. Vertices $I_1, ..I_m$ represent references to schemata $S_{I1}, ..S_{Im}$, vertices $D_1, ..D_n$ represent schema declarations, and vertices $P_1, .., P_o$ represent predicate primes of the specification. In this figure sequential-control arcs are dashed, AND-control arcs are marked by their logical operator.

iii. $V_{end} \subset V$ where $V_{end} \cap V_{pr} = \varnothing$ is a set of vertices called end vertices ($V_{start} \cap V_{end} = \varnothing$).

The vertex $v_t \in V$ is called *totality vertex*. It is a unique vertex such that the *in-degree*$(v_t) = 0$. Any arc $(v_1, v_2) \in A_c$ (referred to as $(v_1, v_2)_c$) is called a sequential-control arc, any arc $(v_1, v_2) \in A_{and}$ (referred to as $(v_1, v_2)_\wedge$) is called an AND-control arc, and any arc $(v_1, v_2) \in A_{or}$ (referred to as $(v_1, v_2)_\vee$) is called an OR-control arc.

Primes that consist of other primes (e.g. in the case of inclusions) contain vertices that refer to those primes. Between the start and end vertices there are those vertices that represent the prime objects of which this higher-order prime is constructed of. Start- and end-vertices thus represent the concept of single entry/exit structures and they are needed to deal with the problem of locality (and scoping). This topic will be discussed in Chap. 5.2.2.

In Z, for instance, a schema block prime is represented by a relatively simply directed graph. Fig. 5.2 demonstrates how a general Z schema block looks like in the SRN representation. Declarations and predicates ($D_1 \,..\, D_n$, and $P_1 \,..\, P_o$) are represented by vertices connected to a start vertex (via AND-control arcs) and to an end vertex (via sequential-control arcs). References to higher-level primes ($S_{I1}..S_{Im}$) are represented by vertices ($I_1 \,..\, I_m$) that are connected to the higher-level primes using AND-control arcs.

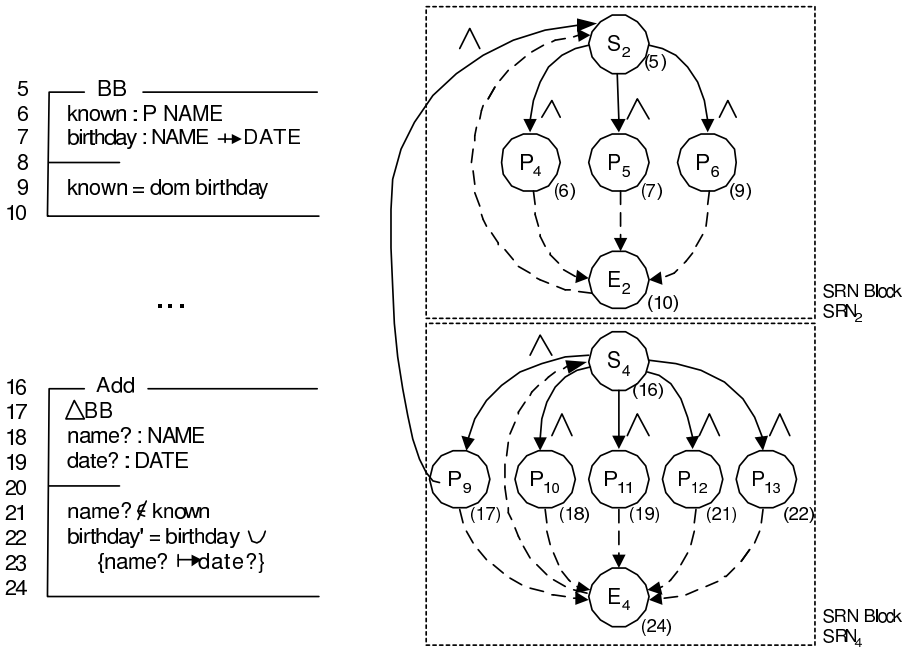


Fig. 5.3: Z specification and SRN representing the birthday book state schema and the *Add* operation schema. Vertices belonging to a set of start and end vertices (SRN blocks) are encapsulated in dotted boxes. For reasons of readability the vertices are annotated by line numbers of the specification source. For instance, vertex P_4 (upper left) represents the predicate prime "$known : \mathbf{P}\ NAME$" at line 6 in the specification, and vertex P_{13} (lower right) represents the predicate prime "$birthday' = birthday \cup \{name? \mapsto date?\}$" (lines 22 and 23 in the specification). Based on this net neighbours, successors and paths can easily be identified.

Most vertices in the net have antecessors and neighbors. If we are only looking at antecessors that are reachable via AND-control arcs, we are looking at so-called AND-antecessors (likewise for OR-antecessors).

Definition 5.5: Antecessor. Let $(V, V_{pr}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$ be an *SRN* representing a specification, and $(v_1, v_2) \in A_c \cup A_{and} \cup A_{or}$. Additionally, let v_3 and v_4 be two vertices in the net ($v_3 \in V$,$v_4 \in V$).

v_1 is referred to as *direct AND-antecessor* of v_2 if $(v_1, v_2) \in A_{and}$. It is a *direct OR-antecessor* if $(v_1, v_2) \in A_{or}$. Otherwise, v_1 is referred to as *direct antecessor* of v_2. v_2 is said to be a *direct successor* of v_1.

If there is a path p from v_3 to v_4, v_3 is referred to as *antecessor* of v_4. If it is true for every arc a of path p that $a \in A_{and}$, then v_3 is referred to as *AND-antecessor* of v_4; if $a \in A_{or}$, v_3 is referred to as *OR-antecessor* of v_4. v_4 is said to be a *successor* of v_3.

To know about vertices' antecessors in the net is important. As will be discussed in Chap. 5.2.2 they tell a lot about structural relationships. Neighbors, on the other hand, indicate that the vertices belong to the same syntactical block.

Definition 5.6: Neighbor. Let $(V, V_{pr}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$ be an *SRN* representing a specification with $(v_1, v_2) \in A_c \cup A_{and} \cup A_{or}$, and $(v_1, v_3) \in A_c \cup A_{and} \cup A_{or}$.

v_2 is said to be an *AND-neighbour* of a vertex v_3 iff $(v_1, v_2) \in A_{and} \wedge (v_1, v_3) \in A_{and}$). Iff $(v_1, v_2) \in A_{or} \wedge (v_1, v_3) \in A_{or}$, v_2 and v_3 are called OR-neighbours.

In the SRN of the birthday book (see Fig. 5.3) vertex S_4 is an AND-antecessor of P_9, P_{10}, P_{11}, P_{12} and P_{13}. However, S_4 is also an AND-antecessor of P_6 as there is a path from S_4 to P_6 (over AND-control arcs from P_9 to S_2).

If neighbours of the vertex are reachable via AND-control arcs, we refer to them as AND-neighbours (which is the same for OR-neighbours). This means that vertices P_9, P_{10}, P_{11} and P_{12} are AND-neighbours of vertex P_{13}.

As can be seen in Fig. 5.3, SRN start and end vertices form specific sub-structures that are called SRN-blocks.

Definition 5.7: SRN block. Let $(V, V_{pr}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$ be an *SRN* of a Z specification. An SRN block SRN_i is a set of vertices $SRN_i \subseteq V$ containing at least a start vertex $v_s \in V_{start}$ and an end vertex $v_e \in V_{end}$, and a non-empty set of prime vertices $V_{pr_i} \subseteq V_{pr}$. For every prime vertex $v_{pr} \in V_{pr_i}$ in the SRN block it holds that v_s is a direct antecessor of v_{pr}, and v_{pr} is a direct antecessor of v_e.

In the above example vertices P_4 to P_6 are in the same SRN block (SRN_2). The reason is that – for all of the vertices P_4, P_5 and P_6 – vertex S_2 is a direct antecessor and vertex E_2 is a direct successor.

5.2.2 Nesting and Scoping

The concept of scope is extensively used in mathematics. Local definitions are introduced in order to state the current meaning of a mathematical symbol. Expressions like "*let* $n : \mathbb{N}$ *be* ..." are commonly understood when reading a section of a text although there are no formal rules that define the scope of such a textual declaration in the whole text.

The idea of a declaration having a scope that is less than the whole program or specification allows the same identifier to be used for different purposes in separate parts (or contexts). The concept of locality was introduced into programming

languages with the rise of block-oriented languages (e.g. Algol60). Not all programming languages adopted the concept of scoping (e.g. Basic or Cobol, see [ML87, p.187ff]), but the concept of locality reduced complexity (at least in some aspects) and permitted more flexibility in inventing names for identifiers.

However, block structures, nesting of expressions and blocks, and ramification introduced complexity on a different level: scope rules for *resolving* references to identifiers are needed in order to define the set of names that may be used at a given point in the program or specification. A conventional scope rule for block oriented programming languages is Rule 5.1 [ML87, p.195]:

Rule 5.1: Simple scope rules.

1. The scope of a declaration includes the block in which it is found but excludes all blocks surrounding it.

2. The scope of a declaration includes all blocks within the block in which it is found but excludes any blocks in which the same identifier is re-declared.

State-based specification languages like VDM and Z also follow the concept of blocks. Thus, scope rules are important, too. In an SRN there is no strict graphical "ordering" of primes, but there is still a syntactical one. Nested expressions and the scope of identifiers[4] are relevant and have to be dealt with. However, this section provides a first and general definition of a scope, namely of the scope of primes. It is solely based on the structure of the net. The scope of identifiers depends on the specification language at hand and will be discussed for Z in Chap. 5.4.

According to Def. 5.7, SRN blocks are used to form structural entities. An SRN block is a container, and it is basically used to define the scope of a prime. There are two ways to interlink SRN blocks in the graph, and this extension affects the scope of primes:

1. Inclusions of SRN blocks (Fig. 5.4(a)). A schema A which is represented by an SRN sub-graph SRN_A with start node S_{A1}, might include another schema B which is represented by an SRN sub-graph SRN_B with start node S_{B1}. In that case there is an AND arc connecting SRN_A with S_{B1}.

2. References to SRN blocks (Fig. 5.4(b)). Primes and sub-graphs in the SRN can be combined, forming complex, nested structures. Two schemata E (represented by sub-graph SRN_E) and F (represented by sub-graph SRN_F) might

[4] Z follows the "classical", simple scope rules as defined in Rule 5.1. According to [Spi89b, p.36] in Z all identifiers (except that of Z schema names) can additionally have nesting of scope.

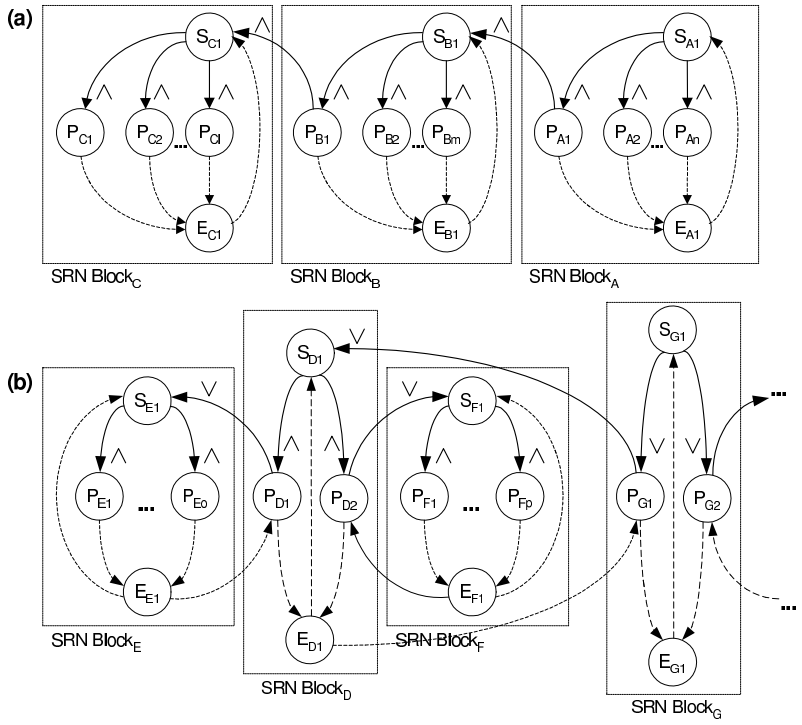


Fig. 5.4: Simple SRN Graph with dotted boxes encapsulating those vertices which are part of an SRN block. Due to the structure of the SRN, two types of nested structures are possible. (a) Inclusions are represented by a single AND-arc that is connected to the included SRN block. Here, *SRN Block*$_C$ is included by *SRN Block*$_B$ due to arc (P_{B1},S_{C1}). (b) References are represented by connecting the referred SRN block to the referring prime via AND and sequential-control arcs. *SRN Block*$_E$ is referenced by *SRN Block*$_D$ via AND-control arc (P_{D1},S_{E1}) and sequential-control arc (E_{E1},P_{D1}).

be combined by a conjunction. In that case a sub-graph SRN_D is introduced, referring to sub-graphs SRN_E and SRN_F. References to sub-graphs are to be found in the net due to two arcs (AND- and sequential-control arcs) connecting the SRN blocks.

SRN blocks, references and inclusions are sufficient for dealing with simple scope rules. Blocks are assigned to SRN-blocks in the net, and these blocks are extended by either references or inclusions. In the case of inclusions the scope is nested; in the case of references the scope (and with it the SRN block) is extended.

Fig. 5.4 demonstrates that sub-graphs might also contain further references and inclusions. This means that nested structures can contain nested sub-structures. As

will be discussed later, this has an impact on the scope of prime objects containing identifier declarations[5].

If a sub-graph A is connected to a sub-graph B, the SRN block of the referenced sub-graph is reachable by the SRN block of the referring sub-graph.

Definition 5.8: Reachability. Let $(V, V_{pr}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$ be an *SRN* representing a specification. An SRN block $SRN_1 \subseteq V$ is said to be *reachable* from an SRN block $SRN_2 \subseteq V$ ($SRN_1 \ \rho \ SRN_2$ for short), iff there exist a start vertex v_{s_1} in SRN_1 ($v_{s_1} \in SRN_1$) and a start vertex v_{s_2} in SRN_2 ($v_{s_2} \in SRN_2$) and a path p from v_{s_2} to v_{s_1}.

If for every arc $a_i \in p$ holds $a_i \in A_{and} \cup A_{or}$, the block is said to be *AND/OR-reachable* ($SRN_1 \ \rho_{AO} \ SRN_2$). Otherwise, if for every arc $a_i \in p$ holds $a_i \in A_{and}$, the block is said to be *AND-reachable* ($SRN_1 \ \rho_A \ SRN_2$). Likewise, if all arcs of p are OR-arcs, a block is said to be *OR-reachable* ($SRN_1 \ \rho_O \ SRN_2$).

The path p is called *reachability path*. The length of the path p is called *reachability distance*. An SRN block SRN_i is said to be element of the reachability path p, if the start vertex v_s of SRN_i ($v_s \in SRN_i$) is element of the path p.

The reachability path is defined as a path between the two start vertices of the involved SRN blocks. In Fig. 5.4 the SRN Block SRN_C is reachable from SRN block SRN_A, as there is a path from start vertex S_{A1} to S_{C1}. The reachability distance of path $p = \langle (S_{A1}, P_{A1}), (P_{A1}, S_{B1}), (S_{B1}, P_{B1}), (P_{B1}, S_{C1}) \rangle$ is 4.

With the definition of an SRN block (see Def. 5.7), the definition of the scope of a prime vertex is possible. Similarly to the definition of the scope of an identifier in Z, the scope of a prime vertex can be defined as those parts of the net, where the prime can be referred to or used[6].

Definition 5.9: SRN scope. Let $(V, V_{pr}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$ be an *SRN* representing a specification, and let SRN_v be an SRN block containing prime vertex v. The SRN scope $\sigma(v)$ of a vertex v ($v \in V$) is a set of vertices, with

(i) $SRN_v \subseteq \sigma(v)$, and

(ii) $SRN_i \subseteq \sigma(v)$, if SRN_i is an SRN block of *SRN* and ($SRN_v \ \rho_{AO} \ SRN_i$).

[5] The scope of a declared identifier in Z denotes those parts of the specification where this identifier can be referred to or used.

[6] Please notice that identifiers are not considered for the scope of primes. Thus, the nesting of sub-graphs has no influence on the scope of primes.

Fig. 5.4 demonstrates how to identify SRN scopes of vertices. The SRN scope of prime vertex P_{B2} consists of two SRN Blocks, *SRN Block*$_A$ and *SRN Block*$_B$, as the vertex P_{B2} is element of *SRN Block*$_B$, and *SRN Block*$_B$ is AND-reachable from *SRN Block*$_A$.

The SRN is only the structural basis for the transformed specification. The transformation itself depends on the specification language that is used. As a first step the subsequent chapter presents the transformation of Z specifications to the SRN.

5.3 Transformation of Z Specifications

Transforming a Z specification to an SRN is simple, but up to now two clues are still missing. Firstly, there are no rules for detecting primes in Z specifications. Secondly, a Z specification uses quite a lot of mathematical symbols and decorations (like boxes and lines), and mark-up languages (like groff/troff or LaTeX[7]) are used to write down specifications: They ease formatting and printing.

Before transformation rules are presented in this chapter, the subsequent sections explain the influence of LaTeX and introduce simple instructions for detecting Z prime objects.

5.3.1 A Word on LaTeX

A Z specification contains more text than is displayed in the pretty-printed specification (see Fig. 5.5). In fact, these LaTeX literals (also called LaTeX text) are used for reasons of pretty printing *and* control. To ensure that a backward transformation of the SRN representation of the specification is possible, LaTeX text also has to be transformed into the net. As will be discussed later, this transformation has some impact on the definition of the SRN.

Breuer and Bowen [BB95] treat terminals that are not part of the grammar of the Z-specification as directives or comments. Additionally, inline annotations in LaTeX are possible, too. Generally speaking there are:

1. *Comments and inline-annotations.* In general a comment (as defined in [BB95]) is a text which is written outside of a Z-paragraph. There is no way of writing comments into a Z schema box or an unboxed paragraph. In Z it is common to annotate paragraphs by using natural language descriptions (which are type-set by using LaTeX mark-ups). Inline-annotations are LaTeX-comments,

[7] For more information on LaTeX see [GMS94]. Information about groff/troff can be obtained via http://www.gnu.org/software/groff. Page last visited: December 2003.

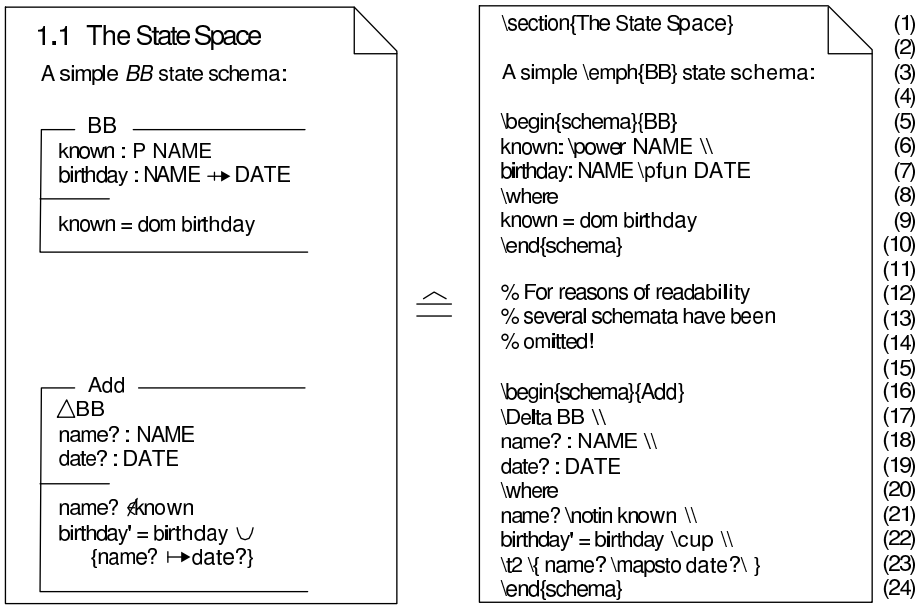


Fig. 5.5: When using LaTeX for type-setting a Z specification, the specification contains more than only Z literals (that are part of Z primes). LaTeX literals and mark-ups are used to format (and pretty-print) the Z specification. On the left side there is a part of the pretty-printed birthday-book specification (commonly referred to as the Z specification) and on the right side, annotated with line numbers, there is the corresponding specification source (commonly referred to as the LaTeX text) is to be found.

starting with a single percent sign "%". It is text that is included in the LaTeX source, but not printed in the Z-specification. The Z specification in Fig. 5.5 contains several comments (lines 1 and 3) and inline-annotations (lines 12 to 14).

2. *Directives.* This is a set of LaTeX commands used to control the output of Z symbols. There are some syntactical restrictions[8], however, as directives can appear inside or outside of a schema box or unboxed paragraph and as they are not displayed in the pretty-printed version of the Z specification.

3. *General LaTeX literals.* This LaTeX text is used to pretty-print the output. It can be, but does not have to be part of a prime object. There are several examples for this situation. Line 23 in Fig. 5.5 has an extra tabulator "$\backslash t2$..." to indent the line. This literal is part of a prime object consisting of lines

[8] A list of Z directives and Z mark-ups can be found at: http://www-users.cs.york.ac.uk/ ian/cadiz/latexmarkup.html. Page last visited: December 2003.

22 and 23. Another example is the LaTeX literal "*where*" in line 20 which draws the line separating the declaration part from the axiomatic part. In many cases these literals, together with specification primes, form higher-level specification prime objects. The LaTeX literal "*begin*{*schema*}{*Add*}" is such a literal. It is not a prime for itself, but together with primes representing declarations and predicates it forms a higher-level prime object representing a schema operation.

There are several possibilities of how to deal with these literals:

(i) The literals are ignored. However, omitting text means to lose information which has been considered necessary by the writer of the specification. Furthermore, a backward transformation of the net to the specification source would be impeded. The structure of the net and the Z primes alone do not suffice to reconstruct the original LaTeX text.

(ii) Another solution would be to attach the additional text to start-, end- and prime vertices. That would mean to mix plain semantic bearing vertices (primes) up with structure bearing vertices. A prime vertex would be more than a prime – it would also take over the role of a structuring entity.

(iii) A third approach is to introduce new types of vertices. Comments and directives are additional information relevant for reading the specification (layout matters). The rest (general LaTeX literals not part of a prime) is of structural nature (be it for LaTeX or for Z).

To ignore the elements means to lose relevant information (in respect to the original specification source). To attach the literals to existing vertices means to overload the semantics of the vertices. Adding new types seems to be an appropriate solution, but how many types of vertices should be added?

In fact, up to now there are already two classes of vertices in the SRN: structure-generating vertices (start- and end-vertices) and semantic-bearing vertices (prime vertices).

Start- and end-vertices are used to form (sub-)units in the SRN. They can represent LaTeX literals like "*begin*{*schema*}{*BB*}", but cannot cope with all LaTeX literals (e.g "*where*", the line separating a schema box). It seems to be very likely to extend this class of structure preserving vertices by a third type of vertex which represents this additional structural information. A vertex of this type is called *structural vertex*.

There are no vertices in the SRN representing comments or annotations of the specification. Thus, a new class of vertices is created. Vertices which belong to this class are called *comment vertices*.

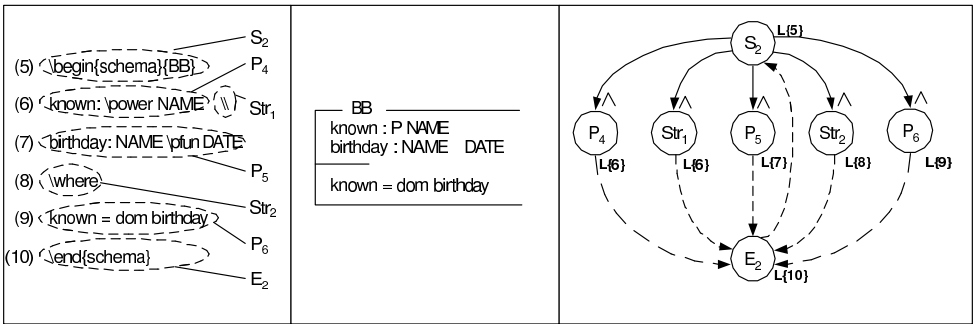


Fig. 5.6: Z primes, Z literals and LaTeX literals are all assigned to the SRN. This figure demonstrates that LaTeX literals are assigned to structural vertices in the net. For reasons of readability line numbers are attached to the vertices in the SRN. Literals, assigned to vertices in the SRN, are encapsulated by dashed boxes.

This separation has several advantages:

1. The general structure of the SRN (sub-graphs encapsulated between start- and end-vertices) remains untouched.

2. Comments, annotations and directives are additional information that can easily be faded in and out when dealing with the net. (That means that it is either possible to focus on the structure and ignore comment vertices, or to focus on the specification text and take the comments into consideration).

Thus, two new types of vertices are introduced. On the one hand, there are comments (annotating the specification), and, on the other hand, there are structuring elements that pertain the readability of the specification.

To summarize, a specification consists of more than pretty-printed prime objects only. In order to deal with the additional information, Def. 5.4 is extended by adding new types of vertices. The eSRN is an SRN which is extended by two types of vertices: vertices representing comments and directives in the LaTeX source code and vertices representing additional structural information. Fig. 5.6 presents an example of an extended SRN. The eSRN is defined as follows:

Definition 5.10: Extended SRN. An extended specification relationship net (also *eSRN* for short) of a specification is a 10-tuple $(V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$ where $(V, V_{pr}, V_{start}, V_{end}, A_C, A_{and}, A_{or}, t)$ is an SRN and $V_{comment}$ and V_{struct} are vertices of V $(V_{comment} \cap V_{struct} = \varnothing)$.

$V_{comment} \subset V$ is a set of vertices called comment vertices; $V_{struct} \subset V$ is a set of vertices called structural vertices. For all vertices $v \in V_{comment} \cup V_{struct}$ it holds that v is not element of $V_{pr} \cup V_{start} \cup V_{end}$.

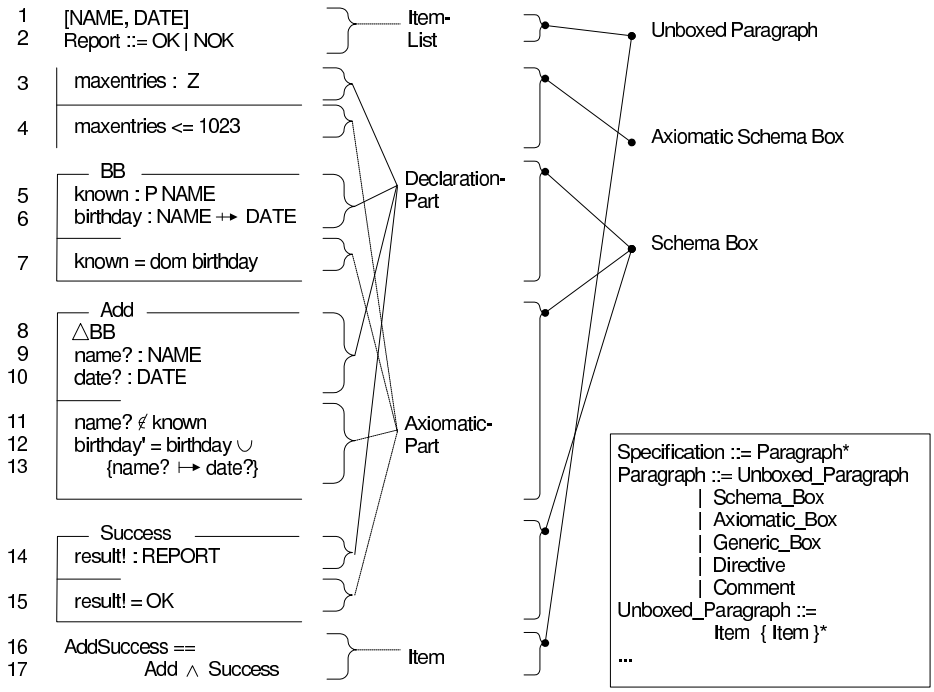


Fig. 5.7: A Z specification consists of a series of Z paragraphs (see [BB95] for details). Generally speaking, there are two types of paragraphs: the first one is undecorated (*Unboxed_Paragraph*), the second one is decorated and is called Z schema box. There are three types of schema boxes (*Schema_Box*, *Axiomatic_Box*, *Generic_Box*), and each of them is split into a declaration part and an optional axiomatic part. Basic prime objects are identified by looking at the non-terminals in the grammar of Z.

With the extension of the SRN mentioned above every literal of a specification's LaTeX text is assigned to the net. The eSRN grasps all prime objects of a specification and provides vertices for syntax-related elements as well as for comments. However, the detection of prime objects depends on the specification language at hand. The subsequent section presents an approach for the identification of primes in Z specifications.

5.3.2 Detecting Primes in Z

Primes are one of *the* semantic bearing entities of a specification. Executing the approach of a specification's transformation, prime objects are assigned to prime vertices in the net. The definition of a prime in Chap. 4.2.1 does not explain how

to identify primes exactly. The reason for this is that the exact definition depends on the specification language at hand.

When transforming a specification to an SRN the choice of the prime objects has a strong influence on the complexity and usefulness of the net. If the mapping is defined too coarse[9], the net is easy to grasp (as only a few prime vertices are in the net), but the chances of reducing the specification are limited (e.g. inter-schema dependencies are vanishing). On the other hand, a too fine granular mapping leads to a rather complex net (which, if the net is used for comprehension activities, is hard to assess). The advantage of a detailed net is that it enables the detection of several interrelationships.

So, what are suitable prime objects for Z specifications? In Chap. 4.2.1 the answer has already been indirectly provided: one has to look for immutable, fundamental units that are merely defined by syntactic rules of the specification language. A look at the Z reference manual [Spi89b] and the Z syntax in [BB95] helps to identify useful primes. The strategy is to look at those non-terminals which form *small*, but not too small, semantic units of a specification.

By following the grammar rules the basic structuring elements are identified in a straight forward manner: They are Z paragraphs which can be unboxed paragraphs and schema boxes. Fig. 5.7 shows an example as well as the basic syntactic elements that Z specifications are constructed upon.

Based on a top-down approach basic prime objects are identified as follows:

(i) The first step takes paragraphs and boxes as basic prime objects. The SRN is then very simple as it contains as many prime vertices in the net as there are paragraphs in the specification. However, this level of granularity does not allow separating pre-condition and post-condition primes (as introduced in Chap. 4.4.1). The prime itself would consist of both pre-condition and post-condition primes at the same time.

(ii) The next step is to split paragraphs into their declaration part and axiomatic part. The argument in step (i) applies again as these elements are too coarse to identify pre-conditions and post-conditions.

(iii) Decomposing these elements furthermore leads to another recommendation of a set of basic prime objects. At this level of granularity pre-condition and post-condition primes can be identified.

[9] According to Chap. 4.2.1 a prime object can be a whole operation schema. On the other, side it can also be a sub-expression within an expression in a Z schema predicate.

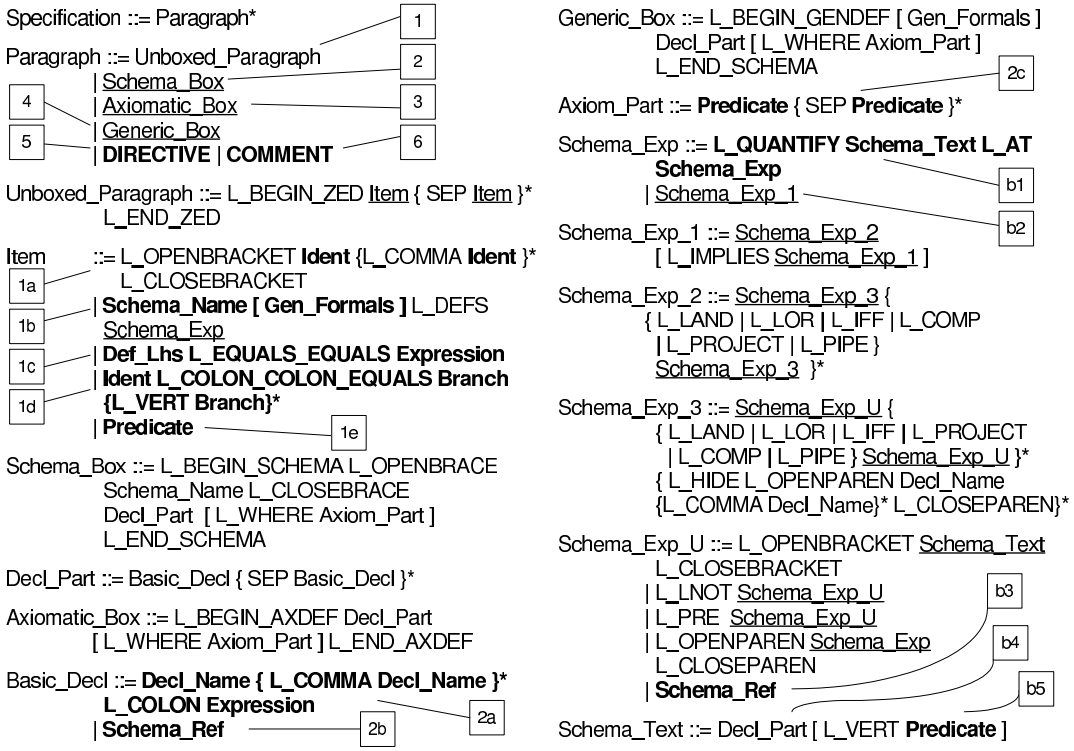


Fig. 5.8: Relevant part of the Z grammar according to the Z reference manual [Spi89b] and the preccx grammar as defined in [BB95]. Upper-case letters indicate terminals, mixed upper- and lower-case letters indicate non-terminals. Non-terminals in bold letters represent prime objects of the specification, underlined non-terminals represent higher-level primes. Literals starting with an "L_" indicate LaTeX literals.

This approach leads to the definition of Rule 5.2 which is based on the Z grammar as defined in [Spi89b, BB95]. Fig. 5.8 presents an annotated version of the relevant parts of the Z grammar.

Rule 5.2: Identification of primes in Z specifications. A Z specification consists of a set of Z paragraphs. A Z-paragraph is either an unboxed Z paragraph 1, a schema box 2, an axiomatic box 3, a generic box 4, a directive 5, or a comment 6. Depending on the corresponding non-terminals, primes are identified as follows (examples refer to the BB-specification in Fig. 5.7):

1. Unboxed Z paragraphs 1. The list of items consists of *given set* declarations, *schema* declarations, *global expressions*, *free type definitions* and *predicates*:

- *Given set declarations* 1a. Each *given set* represents a prime object. In the BB example "*NAME*" and "*DATE*" (line 1) are two prime objects.
- *Schema declarations* 1b. The left-hand side of the declaration is a schema name and represents a prime object. The right-hand side is a schema expression that forms a higher-level prime. "*AddSuccess*" is a prime object, and "$Add \land Success$" is a higher-level prime (lines 16 and 17). The schema expression is further decomposed. It consists of a quantified schema expression b1 or a set of sub-expressions b2. The quantified schema expression forms a semantic unit and is assigned to a prime object. Each sub-expression is treated as a higher-level prime and is decomposed until basic primes like schema references b3, declarations b4, or predicates b5 remain. The expression "$Add \land Success$" is decomposed into "*Add*" and "*Success*", and thus consists of two prime objects.
- *Global Expressions* 1c. The whole expression is treated as one prime object. There is no global expression in the BB-specification. However, an example would be the expression "$square == \lambda\, i : \mathbb{N} \bullet i \cdot i$". It would have been treated as one prime object.
- *Free-type declarations* 1d. Every free type declaration consists of branches (which are either identifiers, or identifiers and expressions). Every free-type declaration forms a prime. "$Report ::= OK \mid NOK$" (line 2) is such a prime object.
- *Predicates* 1e. In Z, a predicate can appear as its own paragraph. This is not the case in the BB-specification. However, a predicate paragraph "$maxentries \geq 0$" might be inserted between lines 4 and 5. In this case, the predicate at hand would be one prime object.

2. Schema 2, axiomatic 3, and generic 4 boxes. They consist of a declaration section and an optional axiomatic section. Looking at these non-terminals leads to the following set of primes:

   - *Declaration Parts.* Each box has a declaration section which might consist of a list of basic declarations. Basic declarations are declaration expressions 2a or references 2b. Each declaration expression is assigned to a prime object. For instance "$maxentries : Z$" (line 3) and "$result! : RESULT$" (line 14) are prime objects. Apart from declarations, references (schema inclusions) might arise in the declaration part. Each reference is represented by a prime. "ΔBB" (line 8) is another prime object in the BB example above.

- *Predicates.* The axiomatic part of a schema box consists of a list of predicates 2c. Here, every predicate is represented by a prime object. "$known = dom\ birthday$" (line 7) or "$report! = OK$" (line 15) are two prime objects of the BB specification.

3. Directives 5 and Comments 6. There are no comments or directives in the BB-example. However, they are directly treated as primes.

□

By applying Rule 5.2 literals forming semantic units of a specification are assigned to prime objects. Schema boxes, axiomatic boxes and generic boxes are decomposed into their declaration and axiomatic part. Every declaration leads to a prime. The same holds for every predicate in the axiomatic part. Comments and directives are directly treated as primes. Unboxed paragraphs are decomposed into items. Items representing predicates, global expressions and generic type definitions are directly assigned to prime objects. Items representing given set declarations and schema expressions are decomposed. Every given set is assigned to a prime object. Schema expressions are decomposed until they are reduced to declarations and predicates.

Primes are identified by a very simple heuristic: looking for the smallest set of Z-literals that (i) form non-terminals of the language definition and that (ii) are commonly referred to when describing or augmenting Z paragraphs. Line numbers are not relevant; what counts are semantic-units that can be assigned to primes. They form the "components" [Spi89b, p.77] that specifications are built upon, and these non-terminals are easy to identify: basic declarations, references, (schema, set, and free type) identifiers, identifiers representing other schema elements and predicates.

It would have been possible to choose a different level of refinement (e.g. staying on the top-down track of decomposition). However, further decomposing the non-terminals is not practical for the following two reasons:

1. The decomposition of predicates (e.g. "$\forall x : \mathbb{P}\, NAME \mid x \subseteq known \bullet \#x \leq maxentries$") leads to several terms and sub-expressions (in the above example the declaration part "$\forall x : \mathbb{P}\, NAME$", the filter "$x \subseteq known$", and the predicate "$\#x \leq maxentries$"). The two terms and the predicate are, if standing alone, too small to form a useful semantic unit. A prime object "$\forall x : \mathbb{P}\, NAME$" definitely forms a syntactic unit, but has too general semantic meaning. For this reason the approach does not decompose nested expressions and handles the whole expression as one prime. The same is true for branches in free-type declarations. An "OK" is almost too small to form a self-containing semantic unit.

2. Developers use Z to model a system by describing static and dynamic behavior via states and operations. Spivey uses the term invariant [Spi89b, p.7] for the relationships which are specified by developers. Invariants are exactly the predicates and expressions identified above. Thus, this choice of basic prime objects seems to be the most appropriate one as these primes are generally used when describing invariants through natural language.

As a further argument rewriters and animation tools for Z are also based on these basic primes. Possum, for instance, looks for declarations and predicates in order to simplify and to transform them to the SUM specification language [HST98] (which can then be animated by executing queries to the specification).

5.3.3 Transformation Rules for Z

The *eSRN* is defined independently of particular specification languages. However, the rules to translate a specification into an *eSRN* in a syntax- and semantic-preserving manner have to be defined in a language dependent manner. This section describes the basic steps that are necessary to transform a Z specification into an eSRN.

According to the grammar of Z (see again Fig. 5.8), there are six types of Z paragraphs: unboxed paragraphs, schema boxes, axiomatic boxes, generic boxes, directives and comments. The creation of an eSRN follows a simple heuristic: Every paragraph in the specification is transformed into an SRN block and is connected to the totality vertex of the net. Primes that represent declarations and predicates are added to that block. Every time a vertex is created the position and text of the prime or literal it represents is attached to it.

However, there are two special situations: Firstly, a prime might represent an inclusion of a schema, and secondly, two schema expressions might be combined by a schema operation. In the first case an AND-control arc is used to connect the referring prime to the referred schema. In the second case an additional SRN block (with a predicate vertex for the left-hand and a predicate vertex for the right-hand side of the expression) is created. The two schema expressions involved are then (recursively) assigned to eSRN subgraphs. Depending on the type of the schema-operation these subgraphs are connected to the predicate vertices. Rule 5.3 describes this approach in more detail.

Several SRN blocks are created during the transformation. The following abbreviations are used to shorten the description of the following procedure:

Definition 5.11: eSRN Operations used in Rule 5.3. The following operations are used when transforming a Z specification to an eSRN. *ID* represents a unique

identifier, SRN represents an SRN subgraph, V represents a vertex in the net, $Prime$ represents a specification prime, OP a schema operator:

- $createSRN : ID \rightarrow SRN$; This operation creates the frame of an SRN block SRN_{ID}. A start vertex S_{ID} and an end vertex E_{ID} are introduced. They are connected by a sequential control arc $(E_{ID}, S_{ID})_C$. It returns the subgraph.

- $addVertex : V \times SRN \nrightarrow V$; This operation takes a vertex V_i and adds it to the SRN block SRN_i (with start vertex S_i and end vertex E_i) via arcs $(S_i, V_i)_\wedge$ and $(V_i, E_i)_c$. It returns the vertex S_i.

- $includeSRN : V \times SRN \nrightarrow V$; This operation takes a vertex V_i and connects it to SRN block SRN_i (with start vertex S_i) via arc $(V_i, S_i)_\wedge$. It returns vertex V_i.

- $referSRN : V \times SRN \nrightarrow V$; This operation takes a vertex V_i and connects it to the SRN block SRN_i (with start vertex S_i and end vertex E_i) via arcs $(V_i, S_i)_\wedge$ and $(E_i, V_i)_c$. It returns vertex V_i.

- $createRefSRN : ID \nrightarrow V$; This operation creates an SRN block SRN_{ID} (via operation $createSRN(ID)$) and a predicate vertex P_r. P_r is connected to this block via operation $referSRN(P_r, SRN_{ID})$ and it is returned.

- $findSRN : Prime \nrightarrow SRN$; This operation returns the SRN subgraph that represents the higher-level prime $Prime$.

- $mergeSUB : (V \times OP \times V \times SRN) \nrightarrow V$; This operation takes two vertices P_1 and P_2 and adds them to a given SRN block SRN_m (with start vertex S_m and end vertex E_m). The following steps are conducted:

  If P_1 is the start vertex of an SRN block SRN_1, a predicate vertex P_1' is created and connected to SRN_1 via $referSRN(P_1', SRN_1)$. Otherwise P_1 is renamed to P_1'.

  If P_2 is a start vertex (of an SRN block SRN_2), a predicate vertex P_2' is created and connected to SRN_2 via $referSRN(P_2', SRN_2)$. Otherwise P_2 is renamed to P_2'.

  Depending on the type of the operator OP, one of the following steps is executed:

  - The operator is a conjunction ($\wedge$), bi-implication ($\Leftrightarrow$), or projection ($\restriction$). Then the prime vertices are connected via arcs $(S_m, P_1')_\wedge$, $(S_m, P_2')_\wedge$, $(P_1', E_m)_c$, $(P_2', E_m)_c$.

- The operator is a disjunction ($\vee$), or an implication ($\Rightarrow$). Then the predicate prime vertices are connected via arcs $(S_m, P'_1)_\vee$, $(S_m, P'_2)_\vee$, $(P'_1, E_m)_c$, $(P'_2, E_m)_c$.
- The operator is a sequential composition operator ($\fatsemi$), or a piping operator ($\gg$). Then the predicate prime vertices are connected via arcs $(S_m, P'_1)_\wedge$, $(P'_1, E_m)_c$, $(S_m, P'_2)_c$, $(P'_1, P'_2)_c$, and $(P'_2, E_m)_c$.

With the operations provided in Def. 5.11 a simple rule for the transformation of a Z specification to an eSRN can be provided:

Rule 5.3: Simple transformation of a Z specification to an eSRN. For every non-empty specification, a totality vertex t is created. The list of paragraphs is decomposed, and, for every paragraph (depending on the non-terminal) one of the following steps is conducted:

1. Unboxed paragraph $\boxed{1}$. The non-terminal is decomposed. An SRN block SRN_p is created via operation $createSRN(p)$, and it is connected to the terminal vertex t by using operation $includeSRN(t, SRN_p)$.

   The terminal L_BEGIN_ZED is assigned to S_p, the terminal L_END_ZED is assigned to E_p. For every terminal SEP a structural vertex P_s is created and added to the SRN block via $addVertex(P_s, SRN_p)$.

   Every non-terminal Item is transformed according to one of the following steps:

   a. Given set declaration $\boxed{1a}$. For every prime object representing a given set a predicate vertex P_i is created and inserted into the SRN block via $addVertex(P_i, SRN_p)$. For every terminal (L_OPENBRACKET, L_COMMA and L_CLOSEBRACKET) a structural vertex P_s is created and added to the SRN via $addVertex(P_s, SRN_p)$.

   b. Schema expression $\boxed{1b}$. A predicate vertex P_i is created and added to the SRN block by operation $addVertex(P_i, SRN_p)$. P_i represents the prime at the left-hand side of the expression.

      For the terminal L_DEFS a structural vertex P_s is created and added to the SRN via $addVertex(P_s, SRN_p)$.

      The non-terminal Schema_Exp (at the right-hand side of the expression) is further decomposed. The following steps are carried out (recursively) until the schema expression is reduced to simple prime objects. The result is a vertex V_{sub} which is added to the SRN block via $addVertex(V_{sub}, SRN_p)$.

i. The non-terminal Schema_Exp consists of a single prime object b1, or a sub-expression b2. If it is a single prime object p_j, a predicate prime P_j is created and assigned to V_{sub}. If it is a sub-expression p, then p is transformed and the result is assigned to V_{sub}. V_{sub} is returned.

ii. The non-terminal Schema_Exp_1 consists of a single sub-expression p_1 (non-terminal Schema_Exp_2). Then p_1 is transformed. The result is assigned to V_{sub}, and V_{sub} is returned.

iii. The non-terminal Schema_Exp_1 consists of two sub-expressions p_1 (non-terminal Schema_Exp_2) and p_2 (non-terminal Schema_Exp_1). Then p_1 and p_2 are transformed. These transformations lead to two vertices $V_{sub,1}$ and $V_{sub,2}$. An SRN block SRN_e is created via $createRefSRN(e)$; the returned vertex is assigned to V_{sub}.
For the terminal L_IMPLIES a syntactical vertex P_s is created and added to the SRN block via $addVertex(P_s, SRN_e)$.
Vertices $V_{sub,1}$ and $V_{sub,2}$ are merged into the actual SRN block SRN_e via $mergeSUB(V_{sub,1}, \Rightarrow, V_{sub,2}, SRN_e)$. V_{sub} is returned.

iv. The non-terminal Schema_Exp_2 consists of a single sub-expression p_1 (non-terminal Schema_Exp_3). Then p_1 is transformed. The result is assigned to V_{sub}, and V_{sub} is returned.

v. The non-terminal Schema_Exp_2 consists of two sub-expressions p_1 and p_2 (non-terminals Schema_Exp_3). Then p_1 and p_2 are transformed. These transformations lead to two vertices $V_{sub,1}$ and $V_{sub,2}$. An SRN block SRN_e is created via $createRefSRN(e)$; the result is assigned to V_{sub}.
For the terminal (L_LAND, L_LOR, L_IFF, L_COMP, L_RESTRICT, or L_PIPE) a syntactical vertex P_s is created and added to the SRN block via $addVertex(P_s, SRN_e)$.
The operation OP is determined, and the subgraphs are added to the SRN block via $mergeSUB(V_{sub,1}, OP, V_{sub,2}, SRN_e)$. V_{sub} is returned.

vi. The non-terminal Schema_Exp_3 consists of a single sub-expression p_1 (non-terminal Schema_Exp_U). Then p_1 is transformed. The result is assigned to V_{sub}, and V_{sub} is returned.

vii. The non-terminal Schema_Exp_3 consists of two sub-expressions p_1 and p_2 (non-terminals Schema_Exp_U). Then p_1 and p_2 are transformed. These transformations lead to two vertices $V_{sub,1}$ and $V_{sub,2}$. An SRN block SRN_e is created via $createRefSRN(e)$; the result is assigned to V_{sub}.
For every terminal which is not part of the sub-expressions, a syntactical vertex P_s is created and added to SRN_e via $addVertex(P_s, SRN_e)$.

The operation OP is determined, and vertices $V_{sub,1}$, and $V_{sub,2}$ are added to the actual SRN block via $mergeSUB(V_{sub,1}, OP, V_{sub,2}, SRN_e)$. V_{sub} is returned.

viii. The non-terminal Schema_Exp_U consists of a single schema reference [b3] referring to prime p_r. A predicate vertex P_r is created and assigned to V_{sub}. The referred SRN block is identified and connected to P_r by $referSRN(P_r, findSRN(p_r))$. V_{sub} is returned.

ix. The non-terminal Schema_Exp_U consists of a schema text expression (non-terminal Schema_Text). This expression consists of declarations, literals and predicates. An SRN block SRN_e is created via operation $createRefSRN(e)$. The result is assigned to SRN_{sub}.
For every basic declaration p_d [b4] a predicate vertex P_d is created and assigned to SRN_e via operation $addVertex(P_d, SRN_e)$. If the basic declaration p_d is a reference to a prime p_r, the referred SRN block is identified and included via $includeSRN(P_d, findSRN(p_r))$.
For every predicate [b5] a predicate vertex P_p is created and assigned to the SRN block via operation $addVertex(P_p, SRN_e)$.
For every terminal that is not part of a prime, a structural vertex P_s is created and assigned to the SRN block via operation $addVertex(P_s, SRN_e)$. Finally, V_{sub} is returned.

x. The non-terminal Schema_Exp_U consists of a schema expression p_s (non-terminal Schema_Exp_U or Schema_Exp) and literals.
The sub-expression p_s is transformed. The result is a vertex $V_{sub,1}$.

* If $V_{sub,1}$ is a single predicate vertex, an SRN block SRN_e is created via operation $createRefSRN(e)$. The result is assigned to V_{sub}. $V_{sub,1}$ is assigned to SRN_e via $addVertex(V_{sub,1}, SRN_e)$.
* Otherwise $V_{sub,1}$ is a vertex referring to an SRN block SRN_{sub}. Then SRN_{sub} is renamed to SRN_e, and $V_{sub,1}$ is assigned to V_{sub}.

For every terminal that is not part of a prime, a structural vertex P_s is created and assigned to the SRN block SRN_{sub} via operation $addVertex(P_s, SRN_e)$. Finally, V_e is returned.

c. Global expression [1c], free-type declaration [1d], or predicate [1e]. For such a prime object a predicate vertex P_i is created and added to the SRN block via $addVertex(P_i, SRN_p)$.

2. Z schema box [2], axiomatic box [3], or generic box [4]. The non-terminal is decomposed. An SRN block SRN_b is created via $createSRN(b)$, and it is connected to the terminal vertex t by using an AND-control arc $(t, S_b)_\wedge$.

Up to the declaration part the terminals and literals are assigned to the start-vertex of SRN_b. The terminal after the axiomatic part is assigned to the end-vertex of SRN_b.

The declaration part is decomposed. For every basic declaration p_d a predicate vertex P_d, and for every terminal SEP a structural vertex P_s is created. They are added to the SRN block via $addVertex(P_d, SRN_b)$ and $addVertex(P_s, SRN_b)$. If the basic declaration is a reference to a prime p_r, the referred SRN block is identified and connected to P_j via $includeSRN(P_d, findSRN(p_r))$.

If there is an Axiomatic_Part, a structural vertex P_s (for terminal L_WHERE) is created and added to the SRN block SRN_b via $addVertex(V_s, SRN_b)$. The axiomatic part is decomposed. For every predicate p_a a predicate vertex P_a, and for every terminal SEP a structural Vertex P_s is created. They are added to the SRN block via $addVertex(P_a, SRN_b)$ and $addVertex(P_s, SRN_b)$.

3. Directive [5] or comment [6]. A comment vertex P_c is created. It is connected to the totality vertex t by using an AND-control arc $(t, P_c)_\wedge$.

□

According to Rule 5.3 every prime and literal of a syntactically correct specification is assigned to the eSRN. By strictly following the grammar-rules, every non-terminal is decomposed to a set of non-terminals (which are finally assigned to prime vertices) and terminals (which are assigned to structural vertices). Every vertex is either added to an SRN block, or directly added to the totality vertex. The eSRN is emerging.

Again, the birthday book specification out of [Spi89b] is taken for illustrative reasons. For reasons of readability comment vertices and structural vertices are omitted in the subsequent figures. Additionally, the *Delete* operation schema of the BB specification is omitted because the transformation steps are the same as those of the *Add* operation schema. In Appendix C.1 there is the full BB specification as well as the transformation to the eSRN[10].

The transformation of schema, axiomatic and generic boxes is straight forward. An SRN block is created, linked to the totality vertex, declarations and predicates are assigned to vertices, and these vertices are linked to the SRN block.

Fig. 5.9 demonstrates how to transform the "Add" operation schema. It includes the state schema *BB* in line 17. According to step 2 (Rule 5.3), an SRN block is

[10] For nearly all Z-examples presented in this work, most of the arcs are AND arcs. The reason for this is that predicates in a Z-schema are, if not otherwise stated, logically combined via an AND operation. This property is, of course, made explicit in the net by using AND-control arcs. However, primes can be combined via disjunction. Thus, OR-control arcs are possible, too.

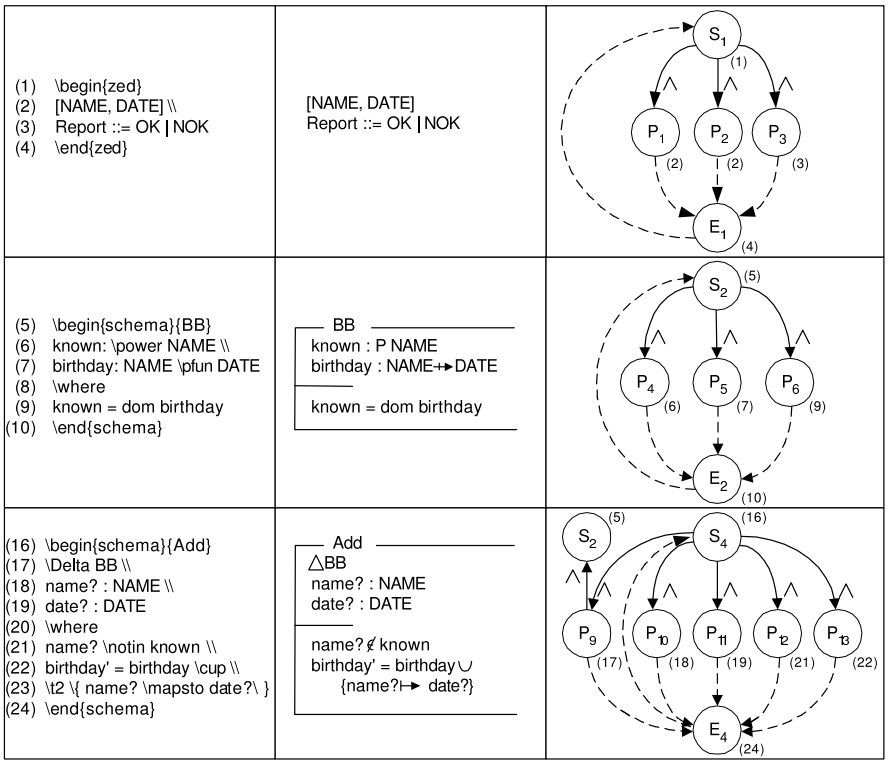


Fig. 5.9: Transformation of three paragraphs of the birthday book specification to the eSRN representation. The columns present the specification source (in LaTeX), the specification and the eSRN. Additionally, every vertex has a line number attached to it. E.g. prime vertex P_4 represents the prime object starting at line (6) in the specification source. For reasons of readability, structural vertices are omitted.

created, and for every basic declaration and predicate, predicate vertices are added to the SRN block. As there is a schema inclusion in the declaration part of the *Add* operation schema, the referred prime is identified (the *BB* state schema), and linked to the referring prime. Thus, the state schema (with start node S_2) is AND-control arc connected to prime object P_9.

The transformation of unboxed paragraphs is, except for one special case – that of a schema expression, again straight-forward. Fig. 5.9 presents a simple example of how to transform an unboxed paragraph (given set and free-type declarations in lines 2 and 3) into an eSRN. An SRN block is created, and the primes are added to the block. P_1 represents the prime object *NAME*, P_2 represents the prime object *DATE*. They are added to the eSRN due to step 1.a in rule 5.3. P_3 represents the prime $Report ::= OK \mid NOK$. It has been added to the eSRN due to step 1.c.

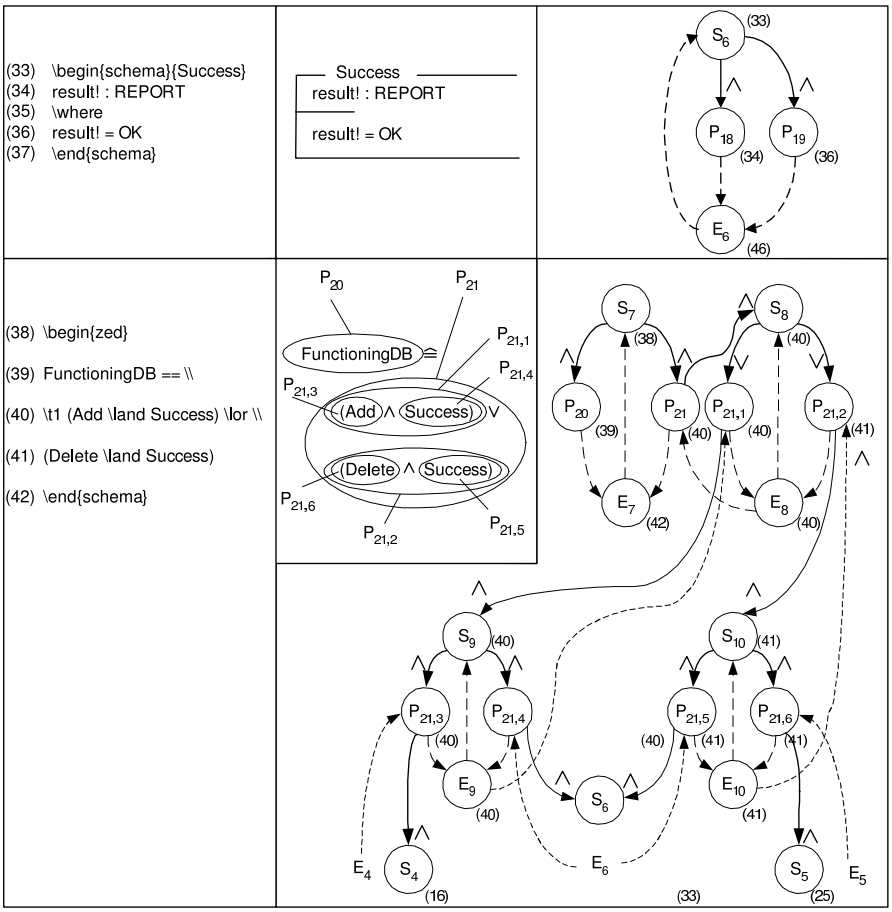


Fig. 5.10: Schemata that are combined via logical operations are encapsulated between nested SRN blocks. The schema *FunctioningDB* is annotated by the corresponding vertices in the eSRN. Additionally, every vertex has a line number attached to it. For reasons of readability, structural vertices are omitted.

Much more complex than operation schemata is the transformation of unboxed paragraphs representing schema expressions. Fig. 5.10 demonstrates how the schema, resulting from the schema expression $FunctioningDB == (Add \land Success) \lor (Delete \land Success)$ (lines 40 and 41 in the specification source), is transformed into the eSRN. As there are several transformation steps to be conducted, Fig. 5.11 summarizes these steps and decomposes the expression into its non-terminals and terminals.

As the paragraph is an unboxed paragraph, step 1 of Rule 5.3 has to be applied. An SRN block SRN_7 is created. The literal "$\backslash begin\{zed\}$" is assigned to S_7, the literal "$\backslash end\{zed\}$" is assigned to E_7. The paragraph consists of one item and does not contain any separators. Step 1.b is applied.

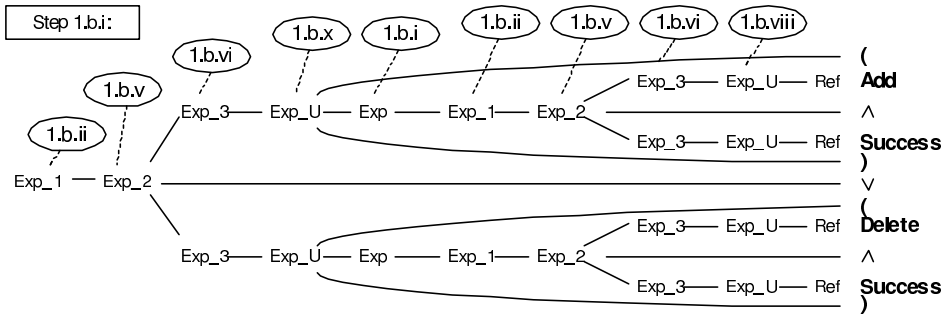


Fig. 5.11: The schema *FunctioningDB* (Fig. 5.10 bottom) consists of schema references that are combined via conjunctions and one disjunction. The schema is transformed into an eSRN by applying rule 5.3. This figure presents the non-terminals and terminals, and, for one schema reference, the steps (beginning with 1.b.i) that are to be conducted.

Predicate vertex P_{20} is created and added to SRN_7. It represents the left hand side of the expression. The literal "==" (terminal L_DEFS) is assigned to a structural vertex and added to SRN_7, too. The same holds for literal "$\backslash t1$". The right-hand side is a single schema expression. Thus, it is transformed and the resulting subgraph (that is referred to from vertex P_{21}) is connected to SRN_7.

The sub-expression is a single subexpression; thus, step 1.b.ii is applied, and the expression is transformed into the eSRN. The expression consists of two subexpressions that are combined by a logical OR operation, applied during step 1.b.v. An SRN block SRN_8 and a vertex P_{21} which refers to SRN_8, is created. The literal "$\backslash lor$" (terminal L_LOR) is assigned to a structural vertex and is added to SRN_8.

The transformation of the sub-expressions are straight-forward. The first sub-expression is reduced (step 1.b.vi) to a single sub-expression and some literals (step 1.b.x). The literals represent the parenthesis, the sub-expression is transformed. The result of this transformation is an SRN block SRN_9. Thus, the literals are added to the block. The same happens to the second sub-expression. The result is SRN block SRN_{10}.

The generation of SRN_9 is as follows: Due to steps 1.b.i and 1.b.ii the sub-expression is reduced to an expression consisting of two sub-expressions and an operator. Step 1.b.v is applied. The literal representing the operator is assigned to a structural vertex, the vertex is added to SRN_9. An SRN block SRN_9 and a vertex $P_{21,1}$ referring to this SRN block are created. This vertex is then included in SRN block SRN_8. However, the result of the transformation of the sub-expressions is included in the SRN block, and, due to steps 1.b.vi and 1.b.viii, it is a single prime vertex $P_{21,3}$. The same holds for the second sub-expression. It is a single schema reference. The result is vertex $P_{21,4}$.

Both vertices are added to SRN block SRN_9. This block is referred to from SRN block SRN_8, which is referred to from SRN_7. The resulting eSRN subgraph can be found in Fig. 5.10.

By following rule 5.3 it is possible to generate an eSRN representation of a syntactically correct specification. Again, the full specification transformation of the birthday book including syntactic vertices can be found in Appendix C.1.2.

5.3.4 Properties of the SRN and the Transformation

Rule 5.3 describes a transformation function which takes a syntactically correct Z specification as input and produces an extended specification relationship net. However, the net is a means to an end. The eSRN is used as a basis for the identification of dependencies and for the generation of specifications' abstractions. It is also of interest whether the specification (in its eSRN form of representation) can be transformed back to its LaTeX form of representation.

This section argues that the transformation, as described in Rule 5.3, is bijective (leading to Theorem 5.7). This implies that a backward transformation exists. The proof makes use of several theorems that are presented hereafter.

The steps presented in Rule 5.3 focus on the special structure of the SRN. Whereas sequential ordering in the LaTeX source text of the specification is neglected, block-building structures and nested expressions are taken into account. When considering properties of the net, strongly connected components are of interest. Strongly connected components are subgraphs, where every vertex in the graph is reachable from all other vertices (and vice-versa):

Definition 5.12: Strongly connected vertices. Let $SRN = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$ be the eSRN of a specification.

Two vertices v_i and v_j are called *strongly connected* ($v_i \sim v_j$), if v_i is reachable from v_j and v_j is reachable from v_i.

$\sim$ defines an equivalence relation on $V \times V$. Sets of vertices satisfying this relation belong to the class of so called *strongly connected components*.

Definition 5.13: Connected components. Let $SRN = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$ be the eSRN of a specification. Let $SRN_{sub} = (V_{sub}, A_{sub})$ be a subgraph of SRN, where $V_{sub} \subseteq V$, $A_{sub} \subseteq (A_c \cup A_{and} \cup A_{or})$.

The sub-graph SRN_{sub} is called *strongly connected*, if it consists of exactly one strongly connected component.

The sub-graph SRN_{sub} is called *connected*, if the corresponding symmetric graph SRN^*_{sub} is strongly connected. The symmetric graph is defined as follows: $SRN^*_{sub} = SRN_{sub} \cup \{(v_i, v_j) \mid (v_j, v_i) \in SRN_{sub}\}$

The steps in Rule 5.3 map block structures of a specification to SRN blocks. The most obvious block structure in Z specifications is that of a boxed Z paragraph. All steps first start to generate a set of SRN blocks, and every prime and literal of this specification block is then embedded in the SRN block. For Z paragraphs it can be observed that nested expressions are dissolved by making use of references and inclusions. Inclusions are represented by AND-arcs connecting a vertex of the referred SRN block to a vertex of the referring SRN block. The involved SRN blocks are not strongly connected (as there is no arc that connects the SRN blocks the other way round). References lead to strongly connected SRN blocks. In fact, for every two (referred) specification blocks a new block is introduced (which refers to those blocks).

A syntactically correct specification consists of a list of specification primes, comments, directives and literals. If there is no risk of misunderstanding, they are summarized under the term *specification elements* in the sequel.

What is even more important is the fact that the steps presented in Rule 5.3 lead to a transformation function that is total:

Theorem 5.1: For every literal in a syntactically correct specification Ψ there is a vertex in the eSRN Υ. The transformation $\mathcal{TR}$, as defined in Rule 5.3, is total ($\mathcal{TR} : \Psi \rightarrow \Upsilon$).

Proof: The proof is divided into two parts: (i) showing that every literal of Ψ is either assigned to a prime object, a structural object, a comment, or directive, and (ii) showing that every such object (generated out of Ψ) is then assigned to a set of vertices in Υ.

ad (i). Section 5.3.2 presented an approach for the identification of primes in Z specifications. A literal in the specification is either element of a Z prime object, or it is a comment, structural or directive literal. Suppose that there exists a literal in Ψ that is not contributing to the non-terminal of a Z paragraph. This can only happen if Ψ is syntactically not correct. In this case the basic assumption is violated because the literal is neither identified as a prime object, nor as a comment, structural or directive literal.

ad (ii). According to Rule 5.3, every non-terminal of a Z paragraph is reduced to a set of primes and literals (step 1 or 2) and directives and comments (step 3).

Unboxed paragraphs are decomposed by steps 1.a to 1.c. Every prime object is assigned to a prime vertex in step 1.a, and every literal is assigned to a structural vertex. Step 1.b recursively decomposes schema expressions. Primes of the schema expression are assigned to the net by steps 1.b.i (quantified schema expression), 1.b.viii (schema references) and 1.b.ix (declarations and predicates). Thus,

all primes are assigned to the net. Composed expressions might contain literals which are not element of a prime object. These literals are assigned to the net by steps 1.b.iii (Schema_Exp_1), 1.b.v (Schema_Exp_2), 1.b.vii (Schema_Exp_3), 1.b.ix (Schema_Exp_Text) and finally 1.b.x (Schema_Exp_U). Thus, all literals of a schema expression that are not part of a prime are assigned to the net. Step 1.c maps expressions, declarations and predicates to the net, and every literal that is element of these non-terminals is part of the prime object.

Schema, axiomatic and generic boxes are transformed by step 2. Again, every prime object is assigned to a prime vertex in the net, and every literal which is not element of a prime is assigned to a structural vertex. Finally, for every directive and for every comment in the specification a vertex is introduced.

Thus, all rules of the Z grammar are handled by the above described steps, and every non-terminal and terminal is assigned to the SRN. However, suppose that there is an element that is not assigned to a vertex in Υ. This means that it is neither a prime object, nor a structural object, nor a comment object, nor a directive. This, in consequence, means that Ψ is syntactically not correct. The basic assumption would be violated. Thus, every object gets its vertex in Υ. The transformation is total.

□

During the transformation phase a set of SRN blocks is generated. These blocks have special properties. First of all, the SRN block represents a strongly connected component because all vertices in the block are reachable from each other.

Theorem 5.2: Every SRN block SRN_i of an eSRN Υ representing a syntactically correct specification Ψ is a strongly connected component. This means

$$\forall\, v_1, v_2 : SRN_i \mid SRN_i \subseteq \Upsilon \bullet (v_1 \sim v_2)$$

Proof: The property of strongly connected components implies that all vertices are mutually reachable in a directed graph (V, A) :

$$\forall\, v_1, v_2 : V \bullet \exists\, p : Path \bullet \alpha(p) = v_1 \wedge \omega(p) = v_2$$

Every SRN block consists of at least three vertices (one start vertex, one end vertex and at least one prime vertex). Whenever steps 1 and 2 of Rule 5.3 are applied, operations *createSRN* and *createRefSRN* are executed. These operations

create a start vertex and an end vertex. The end vertex is connected to the start vertex via a sequential-control arc. Thus, the frame of an SRN block is created.

Whenever such an SRN frame is created, the operation is followed by an *addVertex* operation. *createSRN* in step 1 is followed by an *addVertex* in 1.a, 1.b, and 1.c. The same holds for step 2, where at least one vertex representing a basic declaration is added to the frame. And the operation *createRefSRN* (in steps 1.b.iii, 1.b.v, 1.b.vii, 1.b.ix and 1.b.x) is followed by an *addVertex*, too.

addVertex adds a vertex v to the frame. This vertex is connected to the start vertex, and the end vertex is connected to vertex v. The result is that the start vertex is connected to the additional vertex, the additional vertex is connected to the end vertex, and the end vertex is connected to the start vertex. There is a circle in the initial SRN block, and all vertices are mutually reachable.

□

All vertices in the net are directly connected to the totality vertex or to an SRN block. The SRN blocks are also connected to the totality vertex, or they are referred to from SRN blocks (which are directly or indirectly connected to the totality vertex). This means that an eSRN is a connected graph.

Theorem 5.3: Every eSRN Υ representing a syntactically correct specification Ψ is a connected graph.

Proof: Every literal of a specification paragraph is either transformed into a vertex in an SRN block, or it is transformed into a vertex that is directly connected to the totality vertex t. Every SRN block created via *createSRN* is directly connected to the totality vertex, too. Every SRN block created via *createRefSRN* is connected to a predicate vertex that is connected to this SRN block. This predicate vertex is included via *addVertex* and *mergeSUB* operations to referring SRN blocks, and these SRN blocks are finally referred to from an SRN block created with *createSRN*. This guarantees that there is a path p from the totality vertex to the start node of every SRN block. The graph Υ is connected.

Suppose that there is a vertex v in Υ which is not reachable from the totality vertex. In this case it cannot be a comment vertex, or a directive. The reason is that it is not connected to the totality vertex directly. If it is a prime or structural vertex it has to be a vertex of an SRN block. Because an SRN block is strongly connected, there has to be a path from the start vertex of that SRN block to the vertex v. As there is a path from the totality vertex to every start vertex of an SRN block, this means that there is a path from the totality vertex t to the vertex v. Otherwise the vertex is not a prime or structural vertex, which means that Ψ is syntactically not correct.

□

Additionally, all literals, except those outside a Z schema box, are part of an SRN block.

Theorem 5.4: Every literal that is element of a syntactically correct specification Ψ is part of an SRN block that is element of the eSRN Υ. This holds for all literals except for those representing comments and directives.

Proof: It has already been shown that every literal has a corresponding vertex in Υ. Literals can be part of a prime object. In this case they are directly connected to an SRN block via operation *addVertex*. In all other cases they are assigned to structural vertices, comment or directive vertices. Comments and directives are excluded from SRN blocks (and from this theorem). Structural vertices are directly connected to SRN blocks. Thus, all literals, other than those denoting comments and directives, are element of an SRN block.

□

Literals are not just assigned to vertices in the eSRN. Literals that are closely related (which means that they are element of the same paragraph) are element of strongly connected SRN blocks.

Theorem 5.5: All literals that are element of the same paragraph of a syntactically correct specification Ψ are part of the same SRN block S or of an SRN block T that forms a strongly connected component together with S ($S \sim T$).

Proof: It already has been shown that every literal that is element of a paragraph is assigned to an SRN block. According to transformation steps 1 and 2, every Z paragraph gets its own SRN block. Every vertex representing a literal of this paragraph is added to this SRN block, and is thus strongly connected to the SRN block.

If, according to step 1.b, a higher-level prime is decomposed into several SRN blocks the SRN blocks are getting connected via two opposite arcs (using operation *referSRN*). The SRN blocks are thus strongly connected. Due to this connection, this vertex (which has to be part of an SRN block which is itself a strongly connected component) is element of a strongly connected component.

□

The property of strongly connected SRN blocks will be useful when arguing about structural dependencies in the subsequent section. Next, it is important to show that for every literal in the specification there is a *unique* vertex in the eSRN. This is the case as just one non-terminal or terminal is assigned to the net at a time.

Theorem 5.6: For every specification element (prime or literal that is independent of a prime) of a syntactically correct specification Ψ there is a unique vertex in the SRN Υ. Thus, the transformation is injective ($\mathcal{TR} : \Psi \rightarrowtail \Upsilon$).

Proof: It has already been shown that every literal in a syntactically correct specification Ψ is assigned to a vertex in the eSRN Υ. The transformation into Υ is total. A unique vertex in Υ means that

$$\forall\, x_1, x_2 : \mathit{Element} \mid x_1 \in \Psi;\ x_2 \in \Psi;\ x_1 \neq x_2 \bullet \mathcal{TR}(x_1) \neq \mathcal{TR}(x_2)$$

At a particular point of time the transformation function takes only one non-terminal and replaces it by non-terminals and terminals until a specification element is identified. This object is transformed into the net by creating a new vertex and adding it to the eSRN. There is no step in Rule 5.3 that identifies a prime and assigns it to an already existing vertex in the net. This procedure means that elements in Ψ are identified only once ($x_1 \neq x_2$), and that for every new element a new vertex in Υ is created ($\mathcal{TR}(x_1) \neq \mathcal{TR}(x_2)$). Thus, the transformation is injective.

□

It is important that every specification can be assigned to a unique SRN. However, it is crucial that the SRN representation can be transformed back to the specification in its LaTeX form of representation.

With the theorems discussed above, it is possible to show that the transformation is bijective. This means that there is a backward transformation, and that a transformed specification can be reconstructed without loss of information:

$$\exists\ \mathcal{TR}^{-1} \qquad \wedge \qquad \mathcal{TR}^{-1}(\mathcal{TR}(\Psi)) = \Psi$$

The SRN can be used as a representation replacing the original specification text. The above results are essential for the use of the SRN. This means that every specification can be transformed into an SRN, and that every piece of information in the specification text is assigned to a corresponding vertex in the net. And it is also possible the other way round. Everything interesting in the eSRN can be assigned back to the specification source.

Theorem 5.7: The transformation function $\mathcal{TR}$, transforming a syntactically correct specification Ψ to an eSRN Υ, is bijective ($\mathcal{TR} : \Psi \rightarrowtail\!\!\!\!\!\rightarrow \Upsilon$).

Proof: This theorem implies that (with Υ representing a syntactically correct specification Ψ) the specification source can be reconstructed without loss of information. A bijective function is a function that is both injective and surjective.

Theorem 5.6 already showed that every specification element in Ψ is assigned to a unique vertex in Υ. The transformation is injective.

What is left to show is that the transformation function is surjective ($\mathcal{TR} : \Psi \twoheadrightarrow \Upsilon$). For every vertex in Υ (representing a prime or literal PL) there has to be a distinct specification element (*SpecElement*) in Ψ:

$$\forall\, v : V \mid v \in \Upsilon \setminus \{t\} \ \wedge \ v \in PL \bullet \exists_1\, x : \mathit{SpecElement} \mid x \in \Psi \bullet \mathcal{TR}(x) = v$$

Suppose there is a vertex w that has more than one or no corresponding specification object in Ψ. Two situations can occur:

1. If there are several corresponding objects in Ψ, then Theorem 5.6 would be violated.
2. If there is no corresponding specification element, the net is not an SRN:
   (a) Either the vertex w is isolated (not connected to Υ). This would mean that the net is not an eSRN.
   (b) Or the vertex w is connected to Υ. Then it belongs to an SRN block, or it is connected to the totality vertex. In both cases the vertex is part of a Z paragraph and thus must have been transformed. The only reason for not having been transformed by transformation function $\mathcal{TR}$ is that of an incorrect Ψ.

□

The transformation also preserves the scope of primes in the specification. The scope of a Z prime (see also Rule. 5.1) is defined as follows:

Definition 5.14: Scope of a Z prime. Let Ψ be a syntactically correct Z specification, P and Q be paragraphs of Ψ, and p be prime object that is element of P. The *scope of the prime p*

1. includes the prime p itself,
2. includes those primes q that are element of the same paragraph and
3. includes all paragraphs Q that directly or indirectly refer to paragraph P.

SRN blocks are used to provide the notion of a local scope, and referred/included SRN blocks provide the notion of nested scope. With this it can be shown that the SRN scope equals the scope of primes in Z specifications:

Theorem 5.8: Let p be a prime of the syntactically correct Z specification Ψ, and let v be a vertex in an eSRN Υ representing prime p. Then the SRN scope $\sigma(v)$ equals the scope of p in Ψ.

Proof: In Z the scope is defined by simple scope rules (see Def. 5.14). Firstly, the prime is in the scope of itself. This is also the case in the SRN scope (see Def. 5.9) as the SRN block, that is encapsulating the vertex v, is also element of the scope.

Secondly, the whole paragraph is element of the scope. The paragraph is assigned to an SRN block, and the SRN block is, per definition, element of the SRN scope, too.

Thirdly, it includes all elements of Ψ that are closely related to the prime. It has already been shown that closely related elements in Υ are forming a strongly connected component in the net. For that reason they are reachable from v. The same holds for included elements. They form at least a connected subgraph and are reachable from v, too. The SRN scope can be used to detect the scope of primes in Z specification.

□

Next, in order to discover dependencies in specifications, the *eSRN* graph is enriched by declaration, definition and use information of identifiers attached to the prime vertices. The ASRN emerges.

5.4 Augmenting the eSRN

In the second step of the transformation the *eSRN* is augmented by information to be inferred from the semantic and syntactic definitions of the respective specification language. The particular nature of such dependencies depends again on the specification language under consideration. As with customary compilers, this augmentation step is executed during the transformation into the eSRN. However, for reasons of clearness this process is treated as a distinct augmentation step.

5.4.1 Definitions of an ASRN

An augmented *SRN* is an extension of an *eSRN*, where vertices take attributes describing the "use" of identifiers which belong to these vertices. This "use" can be the declaration or assignment of variables, the use of variables, a type, an input or output channel declaration. A variable is nothing else than a specification literal which is used as a variable in the specification. In the predicate "*known* = dom *birthday*", "*known*", "dom" and "*birthday*" are specification literals, and "*known*" and "*birthday*" are specification literals representing variables.

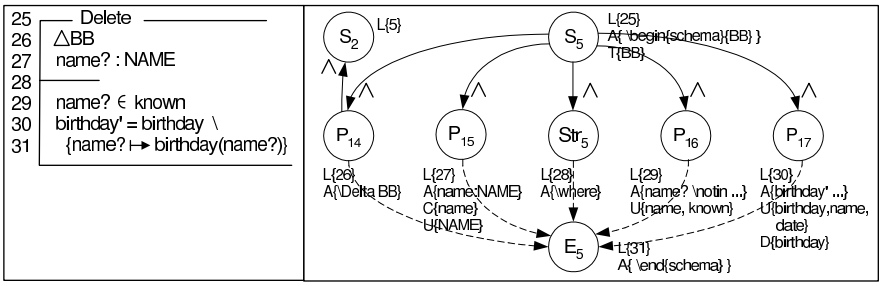
During the transformation step predicate-, start- and end-vertices are augmented. Structural vertices and comment vertices are not augmented. This means that only the *SRN* sub-graph of the *eSRN* is affected. However, as only the *eSRN* represents the full specification source, the following definition and rule are based on the *eSRN*.

Definition 5.15: Augmented SRN. An augmented specification relationship net (ASRN) of a specification is a 7-tuple $(N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ where N_{eSRN} is an extended specification relationship net (V, V_{pr}, V_{start}, V_{end}, V_{struct}, $V_{comment}$, A_c, A_{and}, A_{or}, t), Σ_v is a finite set of symbols called variables, and D, U, T, T_l, C are five total functions with the following semantics:

- $D: V \rightarrow P(\Sigma_v)$. The function $D(v)$ leads to the set of all variables that have a value assignment (are defined) at vertex v.
- $U: V \rightarrow P(\Sigma_v)$. The function $U(v)$ leads to a set of variables that are used at vertex v.
  Thus, D and V serve to make def-/use relationships explicit.
- $T: V \rightarrow P(\Sigma_v)$. The function $T(v)$ (type declaration) leads to the set of all declared variables at vertex v.
- $T_l: V \rightarrow P(\Sigma_v)$. Function $T_l(v)$ (local type declaration) leads to the set of all variables at vertex v that are locally declared.
- $C: V \rightarrow P(\Sigma_v)$. The function $C(v)$ (channel type declaration) leads to a set of values that are syntactically defined (variable declaration) as input or output variables.

The five functions are necessary to cope with the use of identifiers in prime objects:

- $T(v), C(v)$. Syntactically, an identifier is declared in a prime, but the declaration is intended to be used also outside the prime. An "$name? : NAME$" in the declaration section of a schema block is such an example. In this case $name?$ is valid even outside the prime object (but still inside a well defined set of blocks). It has an influence on at least every prime element of the axiomatic part of this schema block.
- $T_l(v)$. An identifer is declared for use in the scope of the prime itself. As an example, an identifier n might be declared as a variable in the following exists-expression: "$\exists\, n : \mathbb{N} \bullet \ldots$". In this case the identifier n is only valid between the boundaries of this prime object.



Prime/Literal	v	$T(v)$	$T_l(v)$	$C(v)$	$U(v)$	$D(v)$
$\backslash begin\{schema\}\{Delete\}$	S_5	*Delete*				
ΔBB	P_{14}					
$name?$: NAME	P_{15}			*name*	*NAME*	
$\backslash where$	Str_5					
$name? \in known$	P_{16}				$name, known$	
$birthday' = birthday \ldots$	P_{17}				$birthday, name, date$	*birthday*
$\backslash end\{schema\}$	E_5					

Tab. 5.1: *Delete* paragraph of the birthday book specification, ASRN representation (including augmented vertices) and table summarizing the functions that are applied to the primes in the operation schema. Line numbers and vertex identifiers refer to the full BB specification as presented in Chap. C.1.2.

- $U(v), D(v)$. An identifier can be used at a specific specification prime, or it can get a value assignment at that prime. In Z, a prime "$counter' = counter + inc?$" uses two identifiers ($counter$, inc), and re-defines the value of $counter$.

For all vertices in the net, the eSRN is augmented by declaration, definition and use information of identifiers. Tab. 5.1 presents the *Delete* operation schema and the values of the functions $D(v)$, $U(v)$, $T(v)$, $T_l(v)$ and $C(v)$. The identifier *birthday* in the last prime object v of the BB-*Delete* schema belongs to the set $U(v)$ and to the set $D(v)$, whereas *name* only belongs to the set $U(v)$ of the respective prime. Beside the position and text that is represented by the vertex in the eSRN, the ASRN provides values for identifiers that are (locally) declared, defined and used at the specification element.

5.4.2 Transformation

As mentioned at the beginning of the previous section, the structure of the extended SRN equals the structure of the SRN. The difference between an SRN and an eSRN is that in an eSRN additional structural and comment vertices are present. These vertices do not declare, define or use variables. Thus, the following augmentation

step works on both graphs. The necessary steps for augmentation are simple. Every vertex is augmented by the type of the use of identifiers.

Rule 5.4: Augmenting an eSRN to create an ASRN.

> For every vertex v representing a specification element e extract the identifiers declared, (re-)defined or used at e. Assign them to the range of the functions $T(v)$, $T_l(v)$, $C(v)$, $D(v)$ and $U(v)$.

At the level of identifiers, nested structures imply that the scope of an identifier changes due to a possible re-declaration of the identifier in one of the referring SRN blocks. What is needed is the definition of a scope that considers this property. Here, the ASRN can be used to refine the notion of the SRN scope (Def. 5.8). A vertex in the ASRN is in the scope of another vertex in respect to an identifier i, if it is element of the same SRN block. If it is element of another SRN block, it is necessary to detect wheter the blocks are reachable in the net. In this case they are at least in the same SRN scope. Due to a nested scope, an identifer could have been re-declared in an SRN block that is element of the reachability path (Def. 5.8). In the latter case there is an SRN scope, but no ASRN scope. Formally, the ASRN scope is defined as follows:

Definition 5.16: ASRN scope. Let $(N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be an *ASRN* of a Z specification, and $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$. Let SRN_v be an SRN block of N_{eSRN} containing vertex v, and let i be an identifer that is declared at vertex v ($i \in T(v) \cup C(v)$).

The *ASRN scope* $\sigma(i, v)$ of identifier i at vertex v is a subset of the SRN scope $\sigma(v)$ of vertex v. A vertex w that is element of SRN block SRN_w is element of the ASRN scope of vertex v with regard to identifier i iff

1. $w \in SRN_w$ and $SRN_w \subseteq \sigma(v)$, and
2. there is no local declaration of identifier i at vertex w ($i \notin T_l(w)$), and
3. SRN_w does not contain a vertex with a declaration of identifier i. Thus, it holds that:
$$\neg \exists v : V \mid v \in SRN_w \bullet i \in T(v) \cup C(v)$$
4. no SRN block on the reachability path p between SRN_v and SRN_w contains a vertex with a re-declaration of identifier i.

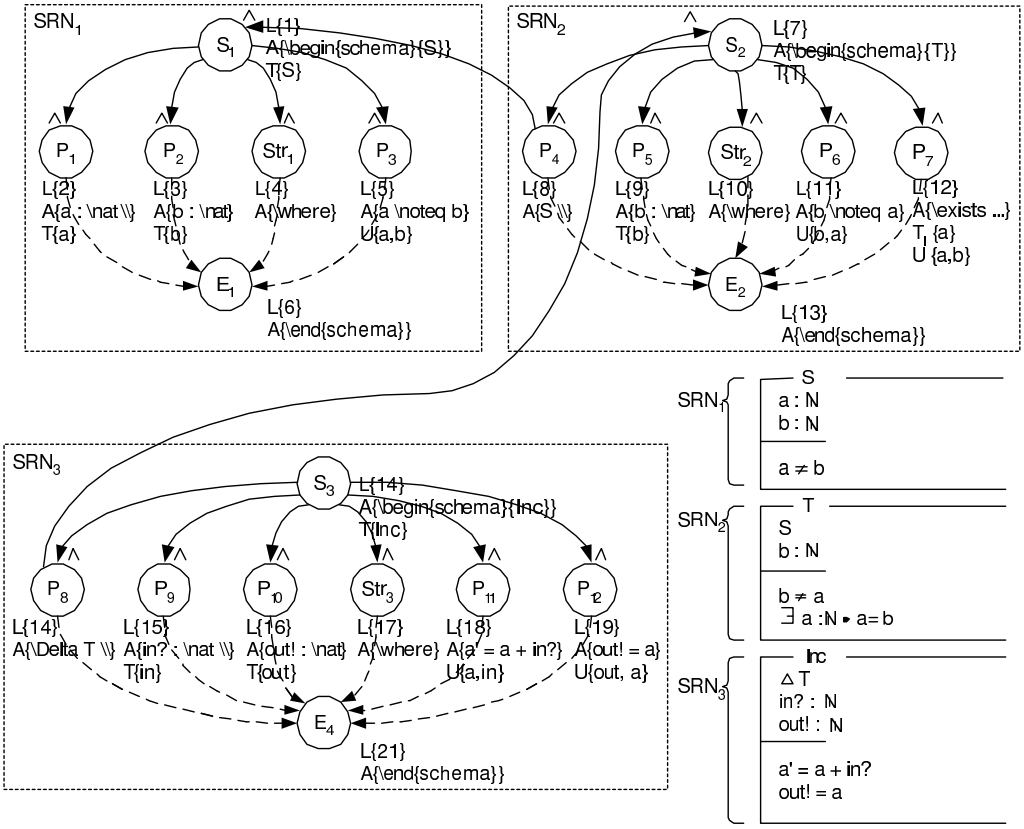


Fig. 5.12: Simple Z specification consisting of two state spaces S and T and an operation schema *Inc*. The operation is used to increment the value of identifier a. This example is used to demonstrate how the ASRN scope of a vertex (in respect to an identifier) is identified.

Fig. 5.12 shows the Z specification of a rather simple example for incrementing the value of an identifier a. However, it covers several cases of references to other schemata. The state space is split into two state schemata, the operation schema does nothing else than adding a value $in?$ to the state space identifier a and returning the value of a.

The SRN scope is defined for a whole SRN block. Thus, SRN blocks SRN_1, SRN_2, SRN_3 are element of the SRN scope of vertex P_2. However, in respect to identifier b, the ASRN scope $\sigma(P_2, b)$ differs from the SRN scope $\sigma(P_2)$. The reason is that identifier b is re-declared in SRN block SRN_2. Thus, every vertex that is element of block SRN_2 is excluded from the ASRN scope.

The same is true for vertices that are element of block SRN_3. In fact, SRN_3 is on the reachability path to block SRN_1, but the same holds for SRN_2 (which has a re-declaration of identifier b and thus violates condition 4 in Def. 5.16).

On the other side, the ASRN scope $\sigma(P1, a)$ is the set of vertices that are element of blocks SRN_1, SRN_2 and SRN_3 – except vertex P_7, as there is a local declaration of identifier a (and thus violates condition 2 in Def. 5.16).

Before demonstrating that the above definition provides a useful approach to identify dependencies in specifications, it has to be shown that the ASRN scope covers the scope of Z primes. The scope of an identifier in Z is defined as follows:

Definition 5.17: Scope of a Z prime in respect to an identifier. Let Ψ be a syntactically correct Z specification, P be a paragraph of Ψ, and p be a prime object that is element of P. Furthermore, let i be an identifier, declared at prime p. The scope of prime p *in respect to identifer* i consists of the scope of prime p (Def. 5.14) but excludes

1. primes q with a local declaration of identifer i, and
2. paragraphs that re-declare identifer i, and
3. all paragraphs that refer to paragraphs that are excluded due to condition 2 or 3.

With that it is possible to show that the ASRN scope (as defined in Def. 5.16) can be used to identify the scope of an identifer i which is element of a specification prime.

Theorem 5.9: Let p be a prime in the syntactically correct Z specification Ψ. Furthermore, let v be a vertex in the eSRN Υ representing prime p, and let z be an identifier declared at vertex v. Then the ASRN scope $\sigma(z, v)$ equals the scope of the identifier z in Ψ at element p.

Proof: It has already been shown that Υ copes with all information of Ψ, and that Υ contains all primes and literals of Ψ. The ASRN is structurally equivalent to Υ; it just makes the information about the declaration, definition and use of identifiers explicit. For that reason it does not matter whether a literal in Ψ or the corresponding vertex in Υ is considered in order to find out the use of an identifier in the specification. Four conditions in Def. 5.16 describe whether a vertex w is in the scope of v or not:

1. Every vertex w that is element of the same SRN block has to be in the SRN scope of v. For a vertex v representing prime p, Theorem 5.8 already demonstrated that the SRN scope of v includes exactly those vertices of the eSRN

that are in the scope of p in Ψ. This means that it includes all elements in Ψ that can refer to p.

2. A vertex w is excluded if it has a local re-declaration of identifier i. This is exactly what happens in the specification due to condition 1 in Def. 5.17.

3. The whole SRN block is excluded if it has a re-declaration of identifier i. This is exactly what condition 2 in the Def. 5.17 calls for.

4. All SRN blocks on the reachability path between the declaration of identifer i and the SRN block containing w are examined. If only one of the blocks contains a re-declaration of identifier i, then w is not element of the scope. The reachability path represents references in the net, and thus this condition equals condition 3 in Def. 5.17.

Both definitions start by looking at all elements (and vertices) that are in the scope of an identifier and remove elements (and vertices) that have re-declarations. As a result, exactly those vertices are eliminated in Υ that are also eliminated in Ψ.

In other words, as the transformation is bijective, the scope of an identifier i can be identified by using the ASRN. It represents all elements in Ψ that are in the scope of that identifier i.

□

There are many applications of ASRNs. In the first place an ASRN can be used to identify dependencies in specifications. This approach is simple and it is based on the identification of paths (reachability conditions) in the eSRN. It is described in the subsequent section. Based on this approach, specification abstractions are deduced. These abstractions are discussed in the next but one section.

5.5 Dependencies in Z Specifications

Based on the definitions of scope (in sections 5.2.2 and 5.4.2) and the notion of control- and data- dependencies, different types of dependencies between vertices in the ASRN can be identified. This section provides three definitions of dependencies in Z specifications which then form the basis for the identification of Z abstractions in the subsequent section.

In Z, identifiers have to be declared before they can be used. This implies that primes which use an identifier i have to be in the scope of another prime which declares that identifier. Assigned to the ASRN, this means that a vertex which uses an identifer i has to be in the ASRN scope of the vertex declaring this identifier i. This dependency is termed *declarational dependency*. It corresponds to the definition

of syntactical dependency of primes (Def. 4.15), where one prime is required in order to keep another prime syntactically correct. Declarational dependency is formally defined as follows:

Definition 5.18: Declarational dependency. Let $(N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be an *ASRN* of a Z specification, and $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$, and u and v be any two vertices of the net $(u \neq v)$. Vertex u is said to be declarational dependent on v, iff

1. u is in the ASRN scope $\sigma(z, v)$, and
2. the identifier z is used (but not re-declared) at vertex u.

$$z \in D(u) \cup U(u) \quad \wedge \quad z \notin T(u) \cup T_l(u) \cup C(u)$$

Declarational dependency between two vertices u and v holds, if an identifier z is declared at vertex v and if the identifier is used at the other vertex u (which is in the scope of v).

Fig. 5.13 demonstrates how to identify declarational dependencies. When looking at prime P_{12} (lower left), the identifiers *name* and *known* are used. P_{12} is declarational dependent on prime P_{10} as the variable *name* is used at P_{12}, defined as input variable at P_{10}, and P_{10} is an AND-neighbor of P_{12} (which means that it is in the same SRN scope).

Furthermore, P_{12} is declarational dependent on P_4 (upper left), as at P_{12} the identifier *known* is used, it is element of SRN block SRN_4 which is in the ASRN scope of SRN block SRN_2. This means that there is a path which consists of AND/OR-control arcs from direct AND-antecessor S_4 to vertex P_4.

Theorem 5.10: The ASRN (based on the eSRN Υ) suffices to identify syntactical dependencies of primes in a syntactically correct Z specification Ψ.

Proof: When looking at Def. 4.15, two aspects are important for syntactical dependency between two primes p and q: Firstly the dependent prime p has to be in the scope of prime q, and, secondly, the prime q is necessary in order to keep p syntactically correct. These two aspects are taken into consideration in Def. 5.18.

1. If a vertex u is in the ASRN scope of z ($u \in \sigma(z, v)$) then the element p (representing u in specification Ψ) is in the scope of element q (which is representing v in Ψ). This has already been shown in Theorem 5.9.

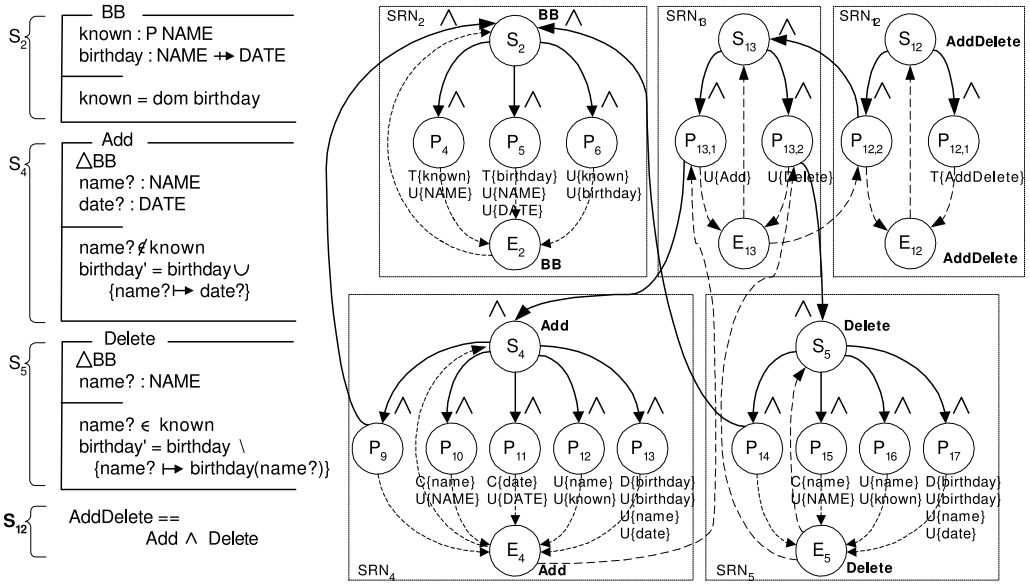


Fig. 5.13: A slightly modified birthday book specification (on the left) and the corresponding ASRN (on the right) in order to demonstrate the identification of Z dependencies. Here, the operation schemata *Add* and *Delete* are combined by using a logical AND operation. SRN blocks $SRN_1 \ldots SRN_5$ are marked by dotted rectangles. Vertices are enriched by their corresponding ASRN information concerning declaration, definition and use of identifiers.

2. Declarational dependency ensures that for every vertex which uses an identifier there is another vertex in the scope that declares (but not re-declares) the identifier. As p and q are unequivocally assigned to the eSRN Υ, and as the backward transformation exists, the same holds for the vertices in the net. Assigned back to the specification Ψ this means that for every prime using the identifier there is another prime (in the scope) that declares the identifier.

□

According to Def. 4.16, post-condition primes are said to be control dependent on the pre-condition primes (if they are in the scope of the pre-condition primes). Pre-condition primes are primes that only contain before-state and input identifiers. Post-condition primes are primes that contain at least one after-state or output identifier. Here the ASRN can be used to identify control dependencies in the net (and thus also in the specification). A predicate vertex v represents a pre-condition prime if it does not contain an identifier that is defined ($D(v) = \varnothing$). On the other

side a vertex u is said to be a post-condition prime if it contains at least one identifier that is re-defined ($D(u) \neq \varnothing$).

If u is in the scope of v then there is control dependency between u and v. However, Chap. 4.4.2 also introduced the notion of control dependency between primes in composed Z schemata. In that case SRN blocks are created, and predicate vertices in the SRN block refer to the referred schemata (which can itself contain references to referred schemata). In such a case it is not enough to look at the scope of a prime v. The SRN block that combines the schemata has to be identified. As related blocks are strongly connected components (see Theorem 5.5), this can be done by looking for start-vertices that are AND/OR-antecessors of vertex v in the net. If there is a path from such an antecessor of vertex v to vertex u, then these vertices are element of the same paragraph. Formally, control dependency is defined as follows:

Definition 5.19: Control dependency in ASRNs. Let $(N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be an *ASRN* of a Z specification, and $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$, and u and v be any two vertices of the net ($u \neq v$). Vertex u is said to be control dependent on v ($u \rightarrow_c v$) , iff

1. $D(v) = \varnothing$ and $D(u) \neq \varnothing$ and $C(v) = \varnothing$ and $T(v) = \varnothing$, and
2. if either
   (a) u is element of the SRN scope $\sigma(v)$, or
   (b) if there is a path p from a start-vertex s which is an AND/OR-antecessor of v to u.

Again Fig. 5.13 can be used to demonstrate the identification of control dependencies. Vertex P_{13} is control dependent on vertex P_{12}, as P_{12} and P_{13} are in the same SRN block (in the same SRN scope), and $D(P_{12}) = \varnothing$ and $D(P_{13}) \neq \varnothing$. Additionally, vertex P_{17} is control dependent on vertex P_{12}, as $D(P_{12}) = \varnothing$ and $D(P_{17}) \neq \varnothing$, and there exists a path p (S_{13}, $P_{13,1}$, S_4, P_{17}) from S_{13} to P_{17}, and S_{13} is an AND-antecessor of P_{13}, which means that the SRN block is AND-reachable (and thus element of the SRN scope).

The ASRN (and Def. 5.19) can be used to identify control dependencies in Z specifications. According to Def. 4.16, a prime p is control dependent on another prime q, if prime q is a pre-condition prime and if prime p is a post-condition prime in the scope of q. As the ASRN can be used to determine scope and pre- and post-conditions, it can also be used to detect control dependency between primes.

Theorem 5.11: The ASRN (based on the eSRN Υ) can be used to identify control dependency between primes p and q in a syntactically correct specification Ψ.

Proof: It has already been shown that the ASRN can be used to identify the scope of a specification element and that the SRN scope equals the scope of primes in the specification. There are two conditions in Def.5.19:

1. Control dependency detection is based on the identification of pre- and post-condition primes in specifications. A syntactical approximation is to look for decorated and undecorated identifiers at specification primes. It has already been shown that it does not matter whether the identifiers are looked for in Ψ or in Υ. In the ASRN the information is explicitly available due to the functions $T(v), C(v), D(v)$ and $U(v)$. By looking for vertices with identifiers that are only used, pre-condition elements are identified.

2. It has already been shown that the scope $\sigma(v)$ can be used to determine the scope of a prime in the specification. However, it is a special case with sequential composition and schema piping. In that case the sequential-control arc is used to connect subgraphs in the ASRN. In Ψ, piping and composition means that one prime can be referred to from the other, but not the other way round. Assigned to Υ this means that it has to be tested whether one vertex is reachable from the other vertex but not the other way round. The ASRN can be used to check these conditions. If there is a path from a start-vertex v (which is an antecessor of the vertex v) to a vertex u, then the vertex u can be referred to (according to Theorem 5.5). As only AND/OR antecessors are considered, the other way round is not possible.

□

According to Def. 4.19, a Z prime is data dependent on another prime, if they share a common identifier in the same scope, and one prime assigns a value to that identifer, and the other prime uses this identifier.

The ASRN can be used to detect data dependencies between vertices in the net (representing two primes in the specification). First the ASRN is used to check whether, at a vertex v, the identifer is re-defined, but not locally re-declared ($z \in D(v) \wedge z \notin T_l(v)$). Then it is used to check whether the second vertex u uses the identifer ($z \in U(u)$). Finally, the net can be used to determine whether they are referring to the same identifier z. Formally, data dependency is defined as follows:

Definition 5.20: Data dependency in ASRNs. Let $(N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be an *ASRN* of a Z specification, with $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct},$

$V_{comment}$, A_c, A_{and}, A_{or}, t), and u and v be any two vertices of the net ($u \neq v$). Vertex u is said to be data dependent on v ($u \rightarrow_d v$), iff

1. there exists at least one variable z so that $z \in U(u)$ and $z \in D(v)$ and $z \notin T_l(v)$, and

2. vertices u and v are declarational dependent (in respect to identifier z) on the same vertex w ($w \in V$). Thus, it holds that:

$$\exists\, z : \Sigma_v;\ u, v : V \mid z \in U(u) \land z \in D(v) \land z \notin T_l(v) \bullet \\ \exists\, w : V \mid z \in T(w) \bullet u \in \sigma(z, w) \ \land \ v \in \sigma(z, w)$$

Fig. 5.13 can also be used to demonstrate how to identify data dependencies by just following the conditions in Def. 5.20. Vertex P_{13} (at the center bottom) is data dependent on P_{17} (lower right), as the identifier *birthday* is used at P_{13} and defined at P_{17}. In addition to that, both vertices are in the ASRN scope $\sigma(P_5, birthday)$ in respect to the identifier *birthday*.

As there is a backward transformation, the ASRN can be used to identify data dependencies in specifications: the specification is transformed into an ASRN, data dependencies between vertices of the net are identified, and these vertices are transformed back to prime objects – indicating data dependencies in the specification.

Theorem 5.12: An ASRN (based on the eSRN Υ) can be used to identify data dependencies between primes p and q of a syntactically correct Z specification Ψ.

Proof: It has already been shown that the ASRN can be used to identify the scope of a prime in respect to an identifier z.

Data dependency is defined (see Def. 4.19) as a relation between two primes in a specification (within the same scope) where these primes share at least a common identifier, and where the value of this identifier is (eventually) changing. To detect data dependencies in an ASRN two conditions have to hold.

1. The vertices have to share at least one common identifier (with a value that is eventually changing). This is checked by looking at the functions U and D in Def. 5.20.

2. The use of the identifier has to happen in the same scope. Again, it has already been shown that the ASRN can be used to identify the scope of an identifier in respect to a specific vertex and that this scope equals the scope of the primes' representation in specification Ψ.

□

Based on the ASRN (and with it the definitions of declarational, control and data dependency) it is possible to identify specification chunks and slices as introduced in Chap. 4.2. The subsequent section provides the definitions.

5.6 Identification of Z Abstractions

Chunks and slices are defined on a specific point of interest in the specification. This point of interest is called abstraction criterion. According to Chap. 4.2.6, the point of interest consists of a prime object, relevant dependencies and a set of literals denoting identifers of the specification. In respect to the ASRN, the abstraction criterion is defined as follows:

Definition 5.21: ASRN abstraction criterion. Let $(N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be an *ASRN* of a syntactically correct Z specification, with $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$.

An abstraction criterion for a specification is a triple (v_p, Θ, Γ), where v_p is a prime vertex ($v_p \in V$) containing the literal l of interest. Furthermore it holds that $\Theta \subseteq \Sigma_v$ is a set of variables defined or used in the ASRN, and Γ is a set of dependencies, $\Gamma \subseteq \{C, D, S\}$, with C denoting control dependency, D denoting data dependency and S denoting declarational dependency.

The notion of chunks is based on the idea of context around a specific point of interest. This point of interest is not restricted to a single point (a single prime) in the specification. Thus, the abstraction criterion of Def. 5.21 is a little bit extended: the prime is replaced by a set of primes. For a specification chunk the point of interest is defined as follows:

Definition 5.22: ASRN chunking criterion. Let $(N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be the *ASRN* of a syntactically correct Z specification, with $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$.

A chunking criterion for a specification is a triple (PL, Θ, Γ), where PL is a set of prime vertices. PL represents primes P of the specification and literals L that also appear within some uniquely identifiable set of prime vertices in the SRN. Furthermore it holds that $\Theta \subseteq \Sigma_v$ is a set of variables defined at literal $l \in L$, and Γ is a set of dependencies, $\Gamma \subseteq \{C, D, S\}$, with C, D, S denoting control-, data-, or declarational dependency respectively.

The chunking criterion according to Def. 5.22 postulates a specification fragment contained within a common context (a set of primes and literals). Analogous to Burnstein's original idea, Def. 5.23 characterizes the *B-chunk* as the immediate context of this specification fragment according to some well defined filter criterion:

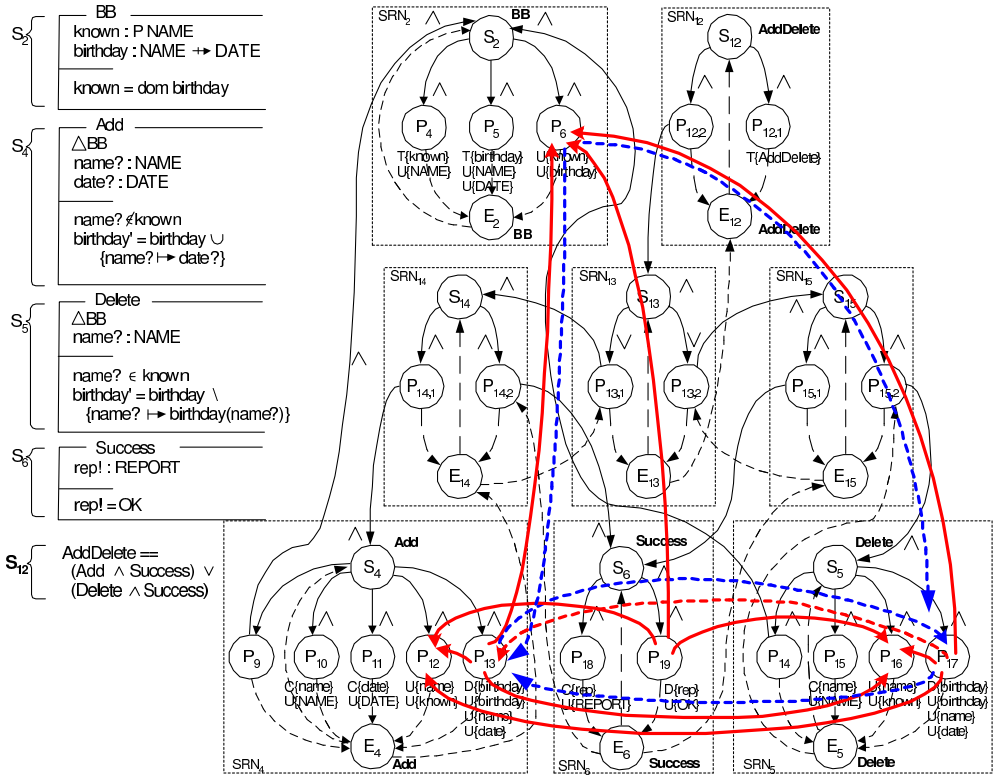


Fig. 5.14: Birthday book specification (on the left) and the corresponding ASRN (on the right) in order to demonstrate the identification of Z abstractions. Here, the operation schemata *Add*, *Success* and *Delete* are combined using a logical AND and OR operation. Control dependencies are visualized via red thick arcs; data dependency is visualized by using blue, thick, dashed arcs.

Definition 5.23: Static Burstein chunk. Let $(N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be the *ASRN* of a Z specification S, with $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$. A static Burnstein chunk, referred to as $BChunk(PL, \Theta, \Gamma)$ of S on chunking criterion (PL, Θ, Γ), $PL \neq \varnothing$ and $\Gamma \neq \varnothing$, is a subset of vertices $V_{BChunk} \subseteq V$, $PL \subseteq V_{BChunk}$ such that for all $v \in V$ holds that vertex $v \in V_{BChunk}$

1. if there exists a data dependency with respect to the variables defined in Θ between vertices in PL and $D \in \Gamma$, or
2. if there exists control dependency between the vertex v and vertices in PL and $C \in \Gamma$, or
3. if there exists declarational dependency between vertex v and vertices in V_{BChunk} and $S \in \Gamma$.

An abstraction criterion focuses on a specific position and a set of literals in the specification. This set of literals can unequivocally be assigned to a set of vertices in the ASRN. The same is true for the chunking criterion which consists of a set of specification elements and every element in this set can be unequivocally assigned to a vertex in the ASRN.

Fig. 5.14 presents a slightly modified specification of the birthday book and the corresponding ASRN. The specification consists of the state space *BB* and three operation schemata: *Add*, *Delete* and *Success*. These schemata are combined in order to make up another schema: *AddDelete*.

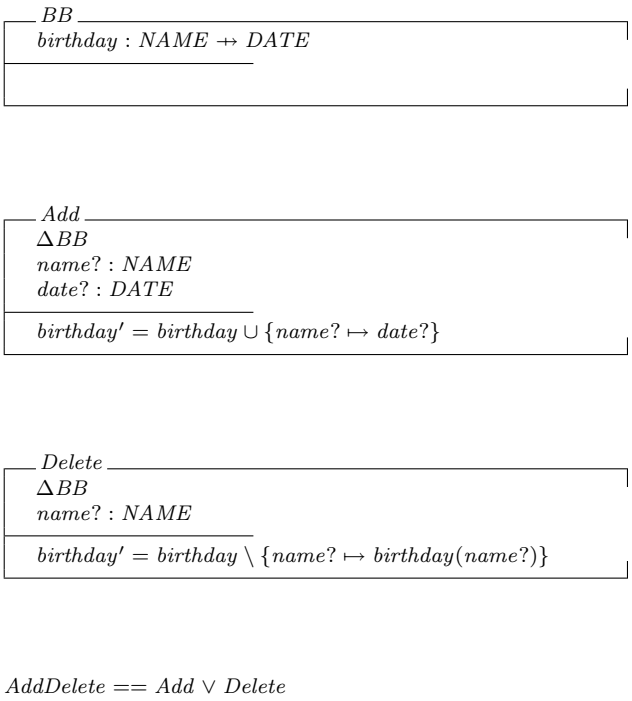
$$
\begin{aligned}
&BB == [known : \mathbb{P}\, NAME;\ birthday : NAME \pfun DATE \mid \\
&Add == [\Delta BB;\ name? : NAME;\ date? : DATE \mid \\
&\qquad name? \notin known;\ birthday' = birthday \cup \{name? \mapsto date?\}] \\
&Delete == [\Delta BB;\ name? : NAME \mid \\
&\qquad name? \in known;\ birthday' = birthday \setminus \{name? \mapsto birthday(name?)\}] \\
&Success == [rep! : Report \mid rep! = OK] \\
&AddDelete == (Add \land Success) \lor (Delete \land Success)
\end{aligned}
$$

Imagine that one is interested to know more about the identifier *birthday* in the *Add* operation schema. As a first step only those primes are of interest that are manipulating the identifier (thus, data dependencies are of interest).

A chunking criterion "$(\{P_{13}\}, \{birthday\}, \{S, D\})$" seems to be appropriate. P_{13} denotes the literal that corresponds to the point of interest: the prime $P_{13} = \boxed{birthday' = birthday \cup \{name? \mapsto date?\}}$. D and S denote that both data dependencies (concerning the identifier *birthday*) and declarational dependencies should be considered. This criterion can be used to calculate the static Burnstein chunk

$$\boxed{BChunk(\{P_{13}\}, \{birthday\}, \{S, D\})}$$

A look at Fig. 5.14 shows the data dependencies that exist in the net. P_{13} is directly data dependent on P_{17} (and vice versa). Thus, both vertices are in the result set. However, P_{13} is not data dependent on P_6 (the dependency works the other way round). As syntactical dependencies are also required by the criterion, all primes that are declarational dependent on vertices P_{13} and P_{17} are included in the result set (for reasons of space not displayed in the figure). The Burnstein chunks corresponds to the following specification:



Quite a lot of primes have been pruned from the specification source. The operation schema *Success* has been removed completely. The same holds for the pre-condition primes in *Add* and *Delete*. With this smaller specification it becomes clear that the identifier *birthday* is used to manage tuples of names and birthdays.

This definition provides the immediate context of the chunking criterion. However, it is not complete in the sense that primes outside of this context might still influence this chunk in an indirect way. Additionally, the chunk might have secondary effects beyond those elements of the specification which it directly affects. Hence, the *full static specification chunk* is proposed next in Def. 5.24. This definition is the transitive closure over the B-chunk, provided in Def. 5.23.

Definition 5.24: Full static specification chunk. Let $(N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be an *ASRN* of a Z specification S, with $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$. A full static specification chunk, referred to as $SChunk(PL, \Theta, \Gamma)$ of S on the chunking criterion (PL, Θ, Γ), is a subset V_{SChunk} of vertices of V ($PL \subseteq V_{SChunk}$), so that for all $v \in V$ it holds that vertex $v \in V_{SChunk}$, iff there is data dependency between at least one vertex $w \in V_{SChunk}$ and v with respect to the variables defined in Θ and $D \in \Gamma$, or there is control dependency

between v and at least one vertex $w \in V_{SChunk}$ and $C \in \Gamma$, or there is declarational dependency between v and any $v \in V_{SChunk}$ and $S \in \Gamma$.

The above application of the Burnstein chunk is useful as it leads to rather small specifications. However, it is often the case that indirect dependencies are necessary in order to understand the interplay of operation schemata in Z. One might want to know which primes influence prime object "$birthday' \ldots$" (P_{13}) in the *Add* schema. At first sight this is a candidate for a Burnstein chunk (thus, generating the immediate context around this prime by including control dependency). $\boxed{BChunk(\{P_{13}\}, \{\}, \{C, S\})}$ leads to:

BB

$known : \mathbb{P}\, NAME$
$birthday : NAME \pfun DATE$

$known = \mathrm{dom}\, birthday$

Add

ΔBB
$name? : NAME$
$date? : DATE$

$name? \notin known$
$birthday' = birthday \setminus \{name? \mapsto date?\}$

Delete

ΔBB
$name? : NAME$

$name? \in known$

$AddDelete == Add \lor Delete$

As there is control dependency between vertex P_{13} and vertices P_{12}, P_{16}, and P_6, these primes are included in the result. One can see that the resulting chunk is indeed only the minimal context around the point of interest. To detect all primes that directly or indirectly contribute to prime P_{13}, a full static specification chunk $\boxed{SChunk(\{P_{13}\}, \{\}, \{C, S\})}$ has to be calculated.

However, the result is a specification containing exactly those primes which are element of the static slice. There are no indirect dependencies in the birthday-book specification.

As declarational dependency is considered, the resulting specification is syntactically correct. On the other side the size of the specification has not been reduced. The reason for this is the fact that the specification is too small, and there is control dependency between all six prime objects in the specification. Chap. 8 presents a more complex specification demonstrating that a full static chunk can be much smaller than the original specification.

As the static chunk extends over the whole specification, it can be directly used for the definition of slices. This replaces the definition of a specification slice given in Def. 4.5. When all types of dependencies are included in the abstraction-criterion of the static chunk, the chunk equals a static specification slice:

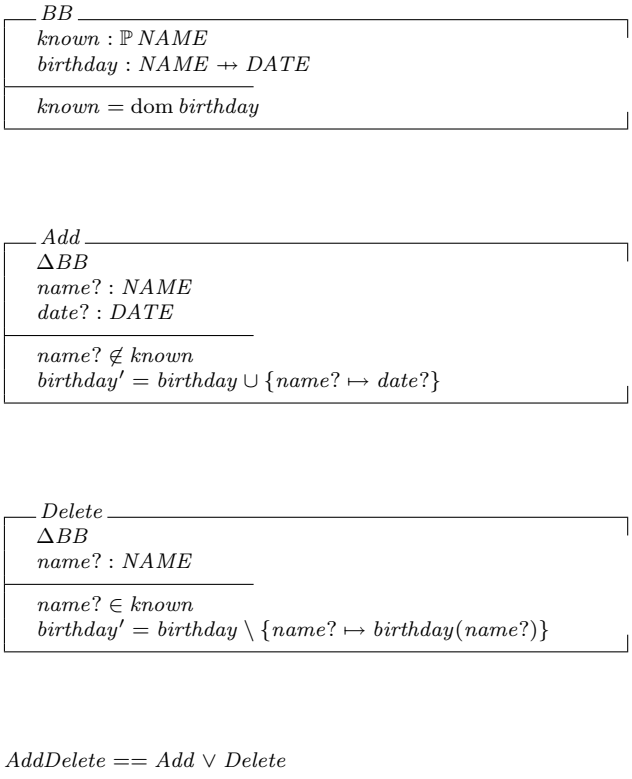
Definition 5.25: Static specification slice. Let $(N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be an *ASRN* of a syntactically correct Z specification, with $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$. Let v_p be a prime vertex containing the literal of interest.

A static slice $SSlice(v_p, \Theta)$ of a specification on a given abstraction criteria (v_p, Θ) equals the static Burnstein chunk $BChunk(\{v_p\}, \Theta, \{D, C, S\})$.

The static specification slice is very useful when looking at a specific point of interest and when considering all dependencies that might arise. When trying to understand the meaning of a prime in a specification, neither data nor control dependencies can be neglected.

Looking at the birthday book specification with respect to identifier *birthday*, the *Add* and the *Delete* operation schemata are of interest. *Success* is not relevant for *birthday*. Thus, when intending to get all relevant primes in respect to a point of interest (e.g. P_{13}), a static specification slice $\boxed{SSlice(P_{13}, \{birthday\})}$ has to be calculated. The result is a syntactically correct specification. It contains prime vertices P_{12}, P_6 and P_{16} as they are control dependent on P_{13}, and prime vertex P_{17} and again P_6, as these vertices are data dependent on vertex P_{13}.

As can be seen in the subsequent example, *Success* is pruned from the specification. No vertex in the SRN block representing the operation schema is directly dependent on the point of interest $\boxed{birthday' = birthday \cup \{name? \mapsto date?\}}$.



The static specification chunk includes all primes that directly depend on the point of interest and all primes that directly contribute to the point of interest. As with full program slices, indirect dependencies in specification are also of interest. Thus, the definition of a full static specification slice is provided:

Definition 5.26: Full static specification slice. Let $(N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be an *ASRN* of a Z specification, with $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$. Let v_p be a prime vertex containing the literal of interest. The full static slice $FSSlice(v_p, \Theta)$ of a specification on a given abstraction criteria (v_p, Θ) is equivalent to the full static specification chunk $SChunk(\{v_p\}, \Theta, \{D, C, S\})$.

With regard to a slicing criterion c the full static specification slice contains all primes that are (directly and indirectly) declarational-, control- and data-dependent on c. This means that it is possible to calculate a specification chunk in respect to this criterion, and to include all types of dependencies. In other words, it holds that a specification slice is nothing else than a special case of a specification chunk.

Containing all types of dependencies it equals the static chunk. However, as all dependencies are included, the above approach does not guarantee that the resulting specification is smaller than the original specification.

With regard to prime vertex P_{13} and identifier *birthday* the full static specification slice is as large as the full specification. Again the reason is the simplicity of the birthday book specification and the fact, that there is no separable component in the specification[11]. Together with static program slicing, the static and full static specification slicing share the problem that a slice is often as large as the original program.

Chunks enable the reader to cope with a rather narrow domain, by highlighting specific parts in a specification. In contrast to chunks, slices focus on a broader domain as they always include all types of dependencies. The advantage is that slices are understandable in this specific domain. As is the case with program code slices, specification slices can become rather big. With densely interwoven specifications, the slice might be identical to the full specification [CR94]. Experiments have shown though [MB03], that substantial reduction of size can be obtained even with relatively small specifications (see also Chap. 8).

5.7 Direct Use of the ASRN

One objective of this work is to solve the problem of how specifications can be made more comprehensible. One proposed solution is to reduce the size and thus omit irrelevant parts of a specification (for a specific problem at hand). Another solution aims at supporting the reconstruction of structure and/or behavior of the original specification (see Chap. 4.1.1). Implicit information is made explicit.

The structure of the net exactly provides this kind of information. It enables the creation of suitable forms of abstractions. Visualization is supported by interacting with the ASRN. From cognition theory ([DDZ96] as cited in [Kad02]) it is known that

> visualization is an act in which an individual establishes a strong connection between an internal construct and something to which access is gained through the senses.

In our cases this "something" is an image – the image of the graph and the opportunity to navigate/to focus on the graph, to play around, to follow different paths. With this it is possible to

[11] In larger systems there are often several interacting objects, and these objects usually form disjoint subsets that do not interact with one another.

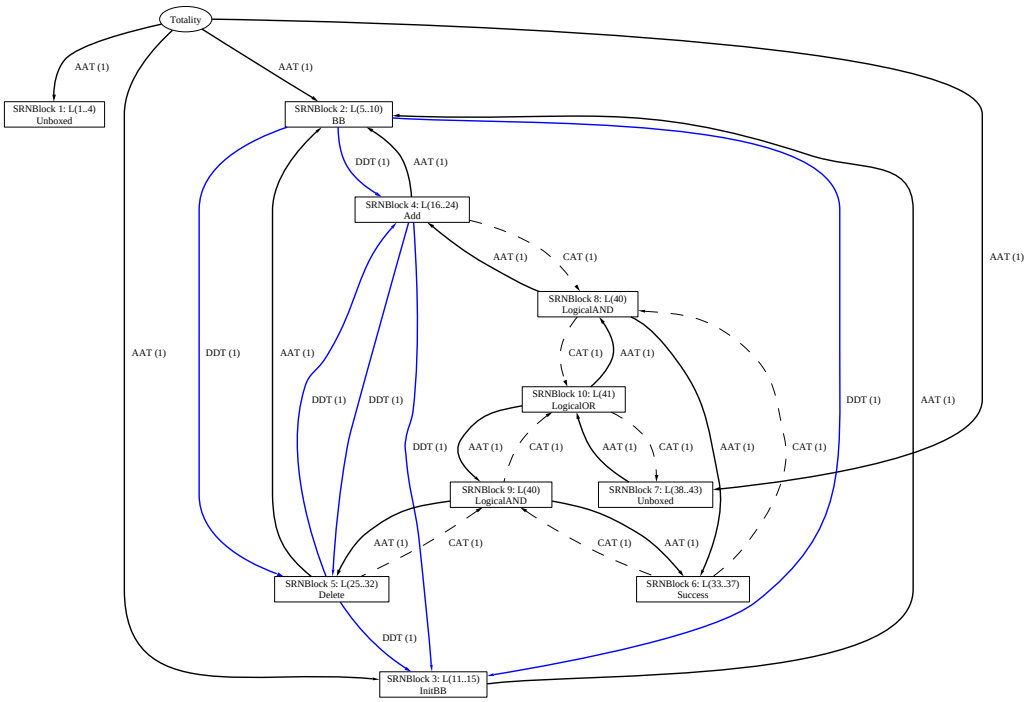


Fig. 5.15: dotty generated image of the eSRN of the birthday book. SRN blocks are represented both via boxes containing an SRN label and the line number(s) in the specification source. Besides, AND-control arcs (bold), sequential-control arcs (dashed) and data dependency arcs (bold blue) are inserted to the net. The layout is generated automatically by using the dot option of dotty's graphviz environment.

- uncover hidden relationships. In the image of the ASRN additional arcs can be introduced, arcs expressing control-, data and declarational dependencies. The previous definitions of the eSRNs and ASRNs do not tell anything about the representation on the screen. So it is quite possible that the graph is used to display only a subset of primes or related primes, display slices and chunks, visually marked SRN blocks, or even partitions and clusters. In Fig. 5.15 the graph is used to display hidden data dependencies between blocks of the specification. It quickly becomes clear that there is data dependency (bold blue arcs) between the schemata *Add*, *Delete*, *BB* and *InitBB*.

- sustain focusing on a specific point of interest. Fig. 5.15 enables to navigate the structure of the specification. Beginning with the totality vertex it quickly gets clear that *Add* and *Delete* are logically AND combined with the *Success* SRN block. In the net the peruser of the image potentially detects that there are quite a lot of dependencies entering and leaving the *Delete* block. Thus,

s/he navigates to this block, eventually examines the vertices in this block and stipulates the calculation of a specification slice.

- emphasize on specific properties of the specification. Such properties might be the number of hidden dependencies, the number of connected and strongly connected sub-graphs and its derived metrics (metrics will be treated in the subsequent chapter in more details). Case studies (see Chap. 8) show that sub-graphs which are correlated by various types of dependencies evolve out of larger specifications. Furthermore, it can be observed that parts of the net are more interwoven than others. This can be used as an indicator for correlation properties. Fig. 5.16 (for reasons of space on the last page of this chapter) presents an overview of the Elevator specification (see Chap. C.3 for an annotated specification source), a specification describing a simple elevator control system. It is noticeable that there are three regions that are not that closely related. The first region (top right) is concerned with schemata opening and closing the door. The second (larger) region in the center of the image deals with all aspects of movements and the last region (bottom left) is concerned with events.

Variations of the graphical representation (the image) of the ASRN are not the topic of this work. However, the prototype used for the calculation of slices and chunks in Chap. 8 provides the possibility to export the ASRN into a *dotty* format[12], and thus indirectly enables the output of a graphical image.

5.8 Summary

This chapter starts with the overall motivation for the transformation of a specification into an augmented net. It explains why an abstract syntax tree is less appropriate for out purpose (the notion of control is not dominant in specification languages), and why the transformation pays off (the net can be used as a means of dependency calculation, visualization, and metrics calculation).

The net is used to cope with syntactic and semantic information. The 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. The SRN is independently defined of a specification language in Chap. 5.2.

[12] *dotty* is a simple toolkit available for Unix and Windows platforms. It enables pretty printing of various forms of graphs described by a rather simple text-based definition language.

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. Thus, Chap. 5.3 defines the eSRN (extended SRN), provides an approach for the detection of primes in Z and presents rules for the transformation of Z specifications (which is type-set in LaTeX) into an eSRN.

Additionally, important properties of the eSRN and the transformation are pointed out:

1. The eSRN can be used to deal with any information that exists in the Z specification.
2. The transformation function of a Z specification into the eSRN is *bijective*. This means that a backward transformation is possible.
3. The scope of Z primes can be identified by just using the eSRN. This means that the net can be used to deal with nested expressions and included schemata.

To ease the identification of dependencies, the eSRN is augmented by declaration, definition and use information of identifiers attached to prime vertices. Chap. 5.4 defines the *Augmented Specification Relationship Net (ASRN)*. The ASRN captures the explicit semantics of the specification. It is shown that the ASRN can be used to identify dependencies in Z specifications.

Based on reachability conditions, Chap. 5.5 extends the idea of dependencies proposed in Chap. 4.4. The ASRN is used to define control, data and declarational dependencies in Z specifications. These dependencies are then used to form the basis for the definition of slices and chunks.

Chap. 5.6 redefines the point of interest and introduces the so-called abstraction criterion. It consists of

1. one prime or a set of primes representing one or more relevant prime objects for the problem at hand,
2. a set of identifiers that are to be taken into consideration, and
3. the number and type of dependencies that are relevant.

Based in this criterion and the notion of control, data and declarational dependencies, static chunks, full static chunks (which are just the transitive closure over the static chunk), static slices and full static slices are defined. It turns out that the full static slice is a special case of a full static chunk. The full static slice is a

full static chunk where all types of dependencies are considered. The chapter closes with some examples and presents some ideas for a further use of the ASRN.

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. Thus, it is a candidate for calculating specification metrics, the topic that Chap. 6 is going to deal with.

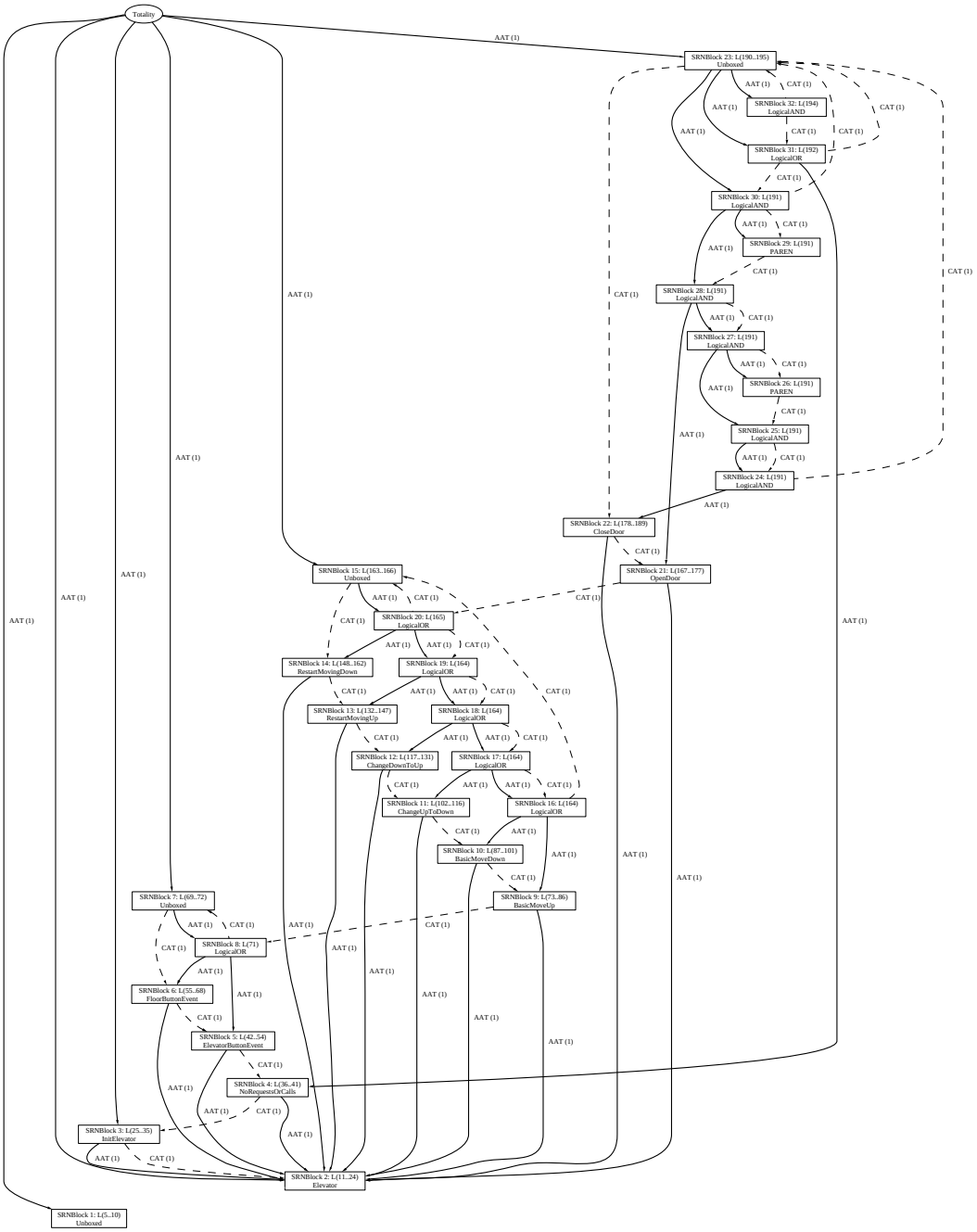


Fig. 5.16: dotty representation of the Elevator specification. SRN blocks are represented via boxes containing an SRN label and the line number in the specification source. AND arcs are bold and sequential-control arcs are visualizes by dashed lines. The layout is automatically generated.

6. SPECIFICATIONS' COMPLEXITY

That many things, having full reference
To one consent, may work contrariously;
As many arrows, loosed several ways,
Come to one mark.

W. Shakespeare (King Henry V., I,2)

Chap. 2 addressed several reasons for the complexity of specifications, and size and hidden dependencies were identified as two of the most crucial ones. Chaps. 4 and 5 claim that it is possible to reduce the complexity by generating specification abstractions. So far it has not been shown that there really is a reduction of complexity. Thus, the goal of this chapter is to provide the basis for such validations. This chapter focuses on the effort which is necessary to measure both complexity in general and complexity of specifications.

There are two possibilities to assess the approach of specification abstractions. Firstly, by describing the effects on complexity via suitable metrics and secondly, by conducting empirical studies. These possibilities are not mutually exclusive. However, even in the case of empirical studies, metrics are necessary for the assessment.

On the search for suitable specification metrics, this chapter starts with popular software metrics. It turns out that size/bulk-based software metrics can be applied to specifications, but that structure-based software metrics need further adaptations.

This chapter suggests to use the ASRN in order to simplify the calculation and to facilitate the application of structure-based metrics. In general, 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.

This chapter is structured as follows. Firstly, software complexity measures are presented. Then, existing specification metrics are discussed. Next, the applicability of procedural metrics to specifications is discussed and, finally, the ASRN is suggested to serve as a basis for the calculation. The chapter closes with a short example demonstrating the simplicity of calculating ASRN-based complexity measures.

6.1 Measuring Complexity

"You cannot control what you cannot measure" is a popular quote from Tom DeMarco in 1982 (reacting to the SW-crisis). Metrics and measurement set up a basis for the controlled software development process.

It is widely agreed that the complexity of software development generally results from interacting effects between the programmer, the program and the programming task. As illustrated in Chap. 2, the same consideration holds for specifications. This implies that complexity factors are not only to be found in the specification source, but that a certain amount of complexity is also inherent to the problem[1] itself.

Measuring complexity is not easy. A lot of metrics have been proposed, but it is still difficult to provide an adequate definition of the term *complexity*. The Webster's Encyclopedic Unabridged Dictionary [Boo96] states that complexity is *the state or quality of being complex*, and the term *complex* is defined as follows:

> **com·plex** (adj.) **1.** composed or interconnected parts; compound; composite. **2.** characterized by a very complicated or involved arrangement of parts, units, etc. **3.** so complicated or intricate as to be hard to understand or deal with. **4.** [...]

Complexity is expressed by the *difficulty* of describing/understanding parts, or, in our case, pieces of software. The Software Engineering Institute provides two definitions of complexity in their Online Glossary[2]:

> (Apparent) The degree to which a system or component has a design or implementation that is difficult to understand and verify [Ins90].
>
> (Inherent) The degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, and the types of data structures [WM87].

Parts that are hard to understand indicate low comprehensibility, low maintainability, low testability and even low reliability and correctness.

It is tempting to define a single value for complexity, and a lot of approaches generate just a single number. However, one gets different complexities from different types of difficulties [Edm99, p.72], and thus complexity cannot be reduced to just one dimension. As stated in [Mil98, p.185]:

[1] Irrespective of the underlying specification language or technique that is used, this problem-inherent complexity will be the same.

[2] SEI Glossary to be found at: http://www.sei.cmu.edu/str/indexes/glossary/complexity.html. Last visited: March 2004.

> "More recent studies of software complexity indicate that there are several basic underlying aspects of software complexity, and that it is unreasonable to expect that any one metric can completely measure the true complexity of a program."

The suggested solution for grasping complexity in its entirety is the combination of different measures. Quite a lot of metrics (see also related bibliography in Chap. B) have been defined and related to each other since the late 1970s. In 1985 Kearney et al. [KST+85] already counted more than 170 experiments investigating the correlation between pairs of complexity measures.

Although many metrics have been proposed, only a few are widely accepted. The objective of this chapter is to identify useful measures for specifications, and the subsequent section starts with an overview and a classification of existing complexity metrics.

6.1.1 Classes of Complexity Metrics

Reliable statements about a system's complexity are formed out of a set of different complexity factors. Many factors have been proposed for measuring complexity, or measuring whatever human beings think of contributing to complexity. There are quite a lot of terms used to describe different types of complexity. Besides the above division into an inherent and apparent view of complexity, there are [FP97, p.245]:

1. *Algorithmic* complexity. Algorithmic complexity is determined by algorithmic requirements and boundaries. This complexity often describes the efficiency of the software.

2. *Conceptual* complexity. Conceptual complexity describes the human costs (time, effort) for creation and understanding. Influencing factors are the logical design, the algorithm, the size and logical depths. Conceptual complexity is often described by means of *design* complexity (sometimes also called *logical* complexity), *computational* complexity, the *information content* and *logical-depth* complexity[3].

It is not always easy to assign one measure to one type of complexity, but it is possible to classify measures with respect to the basis that is used for calculation.

Due to simplicity, two classes of measurement are commonly used since the 1970s: those based on the physical size of the source text and those based on analysis of the flow of data and control [Mil88]. In addition, qualitative measures have been

[3] Here, effective complexity has to be separated from logical depth. The effective complexity of an Mandelbrot algorithm is quite low, but the logical depth is, due to recursion/nesting very high.

developed in order to describe the semantic compactness of components or functions of a system [SMC74] (as cited in [Bal01]).

It is possible to identify three basic classes of metrics: quantity/size based complexity metrics, structure based complexity metrics and semantic based complexity metrics.

M1 *Quantity/Size-based complexity metrics.* Physical size (sometimes also called bulk) has very early been identified as an important factor which determines a system's complexity [She93]. One way to measure the size or bulk of a system (be it a program or a specification) is to *count* the number of occurrences of an item (that is to be specified before). Counting itself looks quite simple, but it is not always clear what to count. The possibilities for counting are manifold as there are quite a lot of different items in the source: number of language statements, decision statements, functions, lines of code, delivered source instructions, number of words or identifiers (just to name some of them). The big advantage of these approaches is that these measures can be expressed in quantitative terms, are easily calculated and well founded due to a huge amount of studies.

M2 *Structure-based complexity metrics.* The general idea behind this class of measures is that systems containing complicated control or data structures are more complex and thus more difficult to comprehend. When looking at the logical structure of a system, the flow of control is of major interest. When looking at data structures, the number of identifiers, their validity and number of references are of interest. Measures belonging to this class of complexity metrics can also be expressed in quantitative terms. However, intermediate forms of representation (mostly trees or flow-graphs) are needed.

M3 *Semantic-based complexity metrics.* When looking at a system (be it a program or a specification), different relations between different components of the system can be detected. Semantic complexity describes the *semantic relationships* between and within those components. They are commonly expressed by the dual properties of *coupling* and *cohesion*. Coupling is a measure for the strength of the inter-component connections, and cohesion is a measure for the mutual affinity of sub-components of a component. Semantic complexity measures cannot be expressed in quantitative terms and are hard to calculate algorithmically.

Size- and structure-based metrics are prevalent and they are used to measure complexity in various fields of application[4]. They often also form the basis for other

[4] They can be and are used for several types of estimations (see [Mil88] for an overview).

estimation models like that of quality or cost/effort [Mil98, p.184]. For the scope of this work we are mainly interested in factors contributing to the complexity of systems. Therefore the subsequent section presents an overview of existing complexity measures.

6.1.2 Popular Complexity Measures

Size-, structure- and semantic-based metrics are often called *product metrics*. This is due to the fact that nearly all measures focus on the final software product: the source and object code. Nevertheless, the term "product" refers to all representational forms that are produced for the required software. This includes specification-, design- and all associated documentation.

For the scope of this section we are interested in *attributes* that are used to indicate software complexity. According to [FP97], typical attributes are length (describing the physical size of the product), structural complexity[5] and functionality. Concerning the separation into three basic classes, popular measures used to quantify these attributes will be discussed in the sequel of this section.

ad M1) Quantity/Size-based metrics have a long tradition. Typical representatives are *Lines of Code (LOC)*, *(thousands of) Delivered Source Instructions (K)DSI*, the *Halstead's Metrics* [Hal77], *Function Points* [Alb79] and the *α-Metric* [KPB98].

LOC seems to be easy to define, but, in fact, there are a number of slightly different[6] definitions to be found in literature [Jon78]. For that reason the approach is not undisputed. Due to these ambiguities results across different projects cannot be compared reasonably. On the other side quite a lot of studies show that LOC, if carefully defined and applied, is a suitable attribute when measuring the complexity of a program (and with it the complexity of a product). In addition to that it is still one of the most important inputs for cost/effort models [Jon86].

In many cases the number of lines of code that are developed is different from the number of lines of code that are delivered. For this reason **(K)DSI** counts those statements of the program which are processed into machine code [FP97, p.248]. Generally speaking, it counts separate statements on the same line of code, but it excludes comment lines. A *DSI* is one statement but should not be mixed up with the measure of the number of *executable statements (ES)*. Unlike executable statements, DSI treats data declarations and headings as source instructions.

[5] Fenton an Pfleeger mention control-flow structure, hierarchical structure and modular structure as examples for attributes contributing to structural complexity.

[6] E.g.: What is to be considered as a line? Are comments or empty lines to be included?

Metric	Definition	Description
Vocabulary η	$\eta = \eta_1 + \eta_2$	total number of unique tokens
Program Length N	$N = N_1 + N_2$	length of implementation
Estimated Length N'	$N' = \eta_1 \cdot log_2\eta_1 + \eta_2 \cdot log_2\eta_2$	estimated program length
Volume V	$V = N \cdot log_2\eta$	measure of storage volume
Difficulty D	$D = (\eta_1 \cdot N_2)/(2\eta_2)$	difficulty to write/understand program
Regularity D	$N = N'/N$	estimates readability

Tab. 6.1: By employing basic measures based on the source text, Halstead defined a set of metrics in order to describe typical characteristics of programs. They also can be used as a basis to calculate the effort and time required to develop a software. Halstead's basic measures are η_1 ... number of unique operators, η_2 ... number of unique operands, N_1 ... total number of used operators, N_2 ... total number of used operands.

Counting lines of code focuses only on one aspect of the underlying product, that of the physical length. In contrast to LOC, **Halstead** [Hal77] devised a set of metrics that apply to several bulk-related aspects of the product. His attempt is to measure *conceptual complexity*[7]. The set of measures (e.g. the volume V of a program or the program length N) is based on the number and use of operators and operands in a program (see Tab. 6.1).

Empirical studies conducted by Halstead show that the values of LOC, N and V are linearly correlated. The metrics are easy to calculate and can be used for all programming languages. However, it is (as Halstead also admits) not always easy to distinguish between operators and operands. Thus, there is the risk of ambiguity in the approach, too. Furthermore, the main focus is on applying the measures onto implementations.

In general, machine-readable documents are missing at the beginning of projects; thus, size based approaches are not that often applied at earlier stages in the SW-development process.

To provide a measure for earlier stages in the development process, Albrecht [Alb79] devised a metric based on the functionality of a product: the **function point metric**. The metric is well-suited for data processing systems, but otherwise less significant. It is primarily a size metric, but the measure is adjusted by various additional factors[8]. The idea is to count basic measures like the number of inputs, outputs, files, references and inquiries. These counts are weighted by factors (derived from empirical observations) and provide the so-called unadjusted function count

[7] Conceptual complexity measures attempt to quantify the amount of difficulty that a programmer will experience in writing or comprehending a piece of code ([Coo82] as cited in [OWE94]).

[8] A detailed list of tools sustaining the calculation of metrics can be found at: http://iac.dtic.mil/dacs/. Page last visited: Dec. 2003.

UFC. The UFC is then adjusted by a technical complexity factor influenced by interface complexity and data communication requirements. These factors allow the UFC to be changed by a factor of ±30%. As the function point approach is not bound to the implementation, nearly every document from which data-processing functions can be deviated can be used as a starting point for the calculation. Several factors are based on experiences gained during projects, but factors change over time. Therefore quite a lot of extensions of the function point method have been developed so far.

Another metric describing the complexity of a program is the so-called $\alpha-$**metric** from Kokol [KPB98]. He measures the information content of a program (and of other textual writings). His approach translates a program into a series of characters (according to a pre-defined translation table relating a character to a 6-bit representation) and transforms the binary representation into a Brownian walk using every 0 bit as a step down and every 1 bit as a step up. It then calculates the root of mean square fluctuation $F(l)$ about the average of the displacement. This approach can be compared to the calculation of the long range correlation in human writings.

In a two-dimensional Brownian walk model $F(l)$ is defined as follows:

$$F(l) = \sqrt{\overline{[\Delta y(l, l_0)]^2} - [\overline{\Delta y(l, l_0)}]^2} \qquad \Delta y(l, l_0) = y(l_0 + l) - y(l_0)$$

When calculating the fluctuation $F(l)$, l is the distance between two points of the walk on the x-axis, l_0 is the initial position on the x-axis where the calculation of $F(l)$ for one pass starts, y is the position of the walk (and thus the distance between the initial position and the current position on the y-axis). The bars indicate the average over all positions l_0.

If the string sequence is uncorrelated, then it holds $F(l) \approx l^{0.5}$. If there are correlations, then it holds that $F(l) \approx l^{\alpha}$ with $\alpha \neq 0.5$. The higher the value of α, the higher the correlation *and* the complexity.

Thus, this complexity measure is related to the information content and the entropy of the analyzed text (source). Higher complexity means more information content and thereafter lesser entropy. Case studies of Kokol et.al demonstrate that the α-metric can be applied to all programming languages for which a translation table is defined. On the other side they also show that there is no correlation between the α-metric and the metrics of Halstead or McCabe (see structure-based metrics below).

ad M2) Structure-based metrics have been introduced as several developers claimed that the size of a program is a relatively unimportant factor in programmer comprehension. Instead the number and types of loops, branches and conditions are

considered to be factors which contribute most to complexity [O'N93, p.204]. Popular measures based on flow of control are Thomas McCabes's *cyclomatic complexity* $v(G)$ [McC76] (and derived extensions) and *knots* of Woodward et.al [WHH79]. Typical measures based on data flow information are Henry and Kafura's *information flow metrics* [HK81] and the $DU(G)$ *metrics* of Tai [Tai84].

McCabe saw no relationship between the length of a module and its complexity. Instead he suggested to take the number of control paths as a "better" indicator, especially as it has a strong correlation to testing effort. The **cyclomatic complexity** $v(G)$ [McC76] of a program is based on its representation as a control flow-graph G. The calculation is based on graph-theoretic concepts, and the basic idea is to measure the maximum number of linearly independent paths[9] through a program [FP97, p.38]. The cyclomatic complexity is extracted by counting the minimum set of paths which can be used to construct all other paths through the graph. It measures aspects of the structural complexity of a program and has originally been devised to estimate the effort of testing. Cyclomatic complexity is not uncontested. Fenton and Pfleeger note that cyclomatic complexity if at all only presents a partial view of complexity [FP97, p.293].

The cyclomatic complexity of a graph with n vertices, e edges and p connected components is calculated as follows:

$$v(G) = e \ - \ n \ + \ 2p$$

If the program consists only of one component, and if there are only binary decisions, then the formula simplifies to

$$v(G) \ = \ \textit{number of decision statements} \ + \ 1$$

In addition to that, a "1" is added to complexity for every logical connective which is used in the decision statement.

The cyclomatic complexity is well-established and has been extended in several ways (see again [Mil88] for a list of references). To broaden the field of application, McCabe and Butler applied the approach to design documents by looking at paths in design trees and design subtrees. They introduced three additional measures: module design complexity $iv(G)$, design complexity S_0 and integration complexity S_1. These metrics are derivatives of the cyclomatic complexity. The approach first reduces the flow-graph by some reduction rules[10] and then calculates the cyclomatic

[9] A set of paths is linearly independent if no path in the set is a linear combination of any other paths in the set [FP97, p.333].

[10] In the flow-graphs only calls to subordinate modules are considered. All other nodes are eliminated. Loops are eliminated, too. The algorithm is described in more detail in [McC89].

complexity of that graph [McC89]. The module design complexity $iv(G)$ is the complexity of the reduced graph. Design complexity is then defined as $S_0 = \sum_{i \in D} iv(G_i)$, and integration complexity is defined as $S_1 = S_0 - n + 1$ (whereby n is the number of modules in the system).

The **knots count metric** is (like cyclomatic complexity) based on control flow information. Woodward et.al suggest to use the number of arc intersections of a program flow graph as an index of structuredness. Every statement and every block of sequentially connected statements is represented by a vertex in a graph. The knot count then represents the number of necessary crossing of directional lines in that graph. This means that the knot count is increased every time the flow of control is crossed. This happens if the language permits a *goto* statement (which allows the flow of control to "jump" directly to a position inside another path in the flow of control). The knot count metric is used to indicate the coding clarity. If the program is fully-structured, the knot-count is zero.

Alternatively the flow and the use of data are used to measure the complexity of a component. Henry and Kafura [HK81] suggest to count the number of local **information flow** entering ($fan - in$, input) and leaving ($fan - out$, output) a procedure. The information flow metric c of a procedure is calculated by the formula

$$c \;=\; [procedure\ length] \cdot [fan - in \cdot fan - out]^2$$

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 [She93, p.41]. As the module's internal complexity is also relevant, Henry and Kafura suggest to measure this internal complexity by counting the lines of code. The $fan - in$ is calculated by adding the number of information flow (terminating at a procedure) to the number of data structures (from which information is retrieved). The $fan - out$ is calculated by counting the number of flows originating at a procedure and adding the result to the number of data-structures (where information is updated).

The **Definition-Use ($DU(G)$) metric** [Tai84] is based on data flow information, but uses a control flow graph G as its basis. The control-flow graph is annotated by definition (d) and use (u) information of identifiers representing data items. The $DU(G)$ value of a structured program is then defined as the maximum number of $(d - u)$ - tuples in the control flow graph.

ad M3) Semantic-based metrics. Up to now it is most difficult to cope with semantic complexity. In the early 1970s Constantine identified five levels of relationship between sets of activities in a program. He called them processing elements and the relationships *associative principles*. Some years later Stevens, Myers and Constantine [SMC74] then extended the list to seven which then de

Cohesion	Description
Coincidental	none of the above
Logical	elements are realizing logically related tasks (one element or the other)
Temporal	elements that are activated at about the same point of time
Procedural	elements that have to be executed in some given order
Communicational	elements operate on the same set of data
Sequential	output of one serves as input for the other
Functional	all parts contribute to one single and specific function

Tab. 6.2: Summary of levels of cohesion (according to [SMC74]) in increasing order. The table describes the level of cohesion between so-called processing elements in a program. The exact meaning of processing elements has been left open in order to permit the classification to be applicable to modules and other elements of design documents.

facto lead to the new standard in describing module cohesion (see Tab. 6.2). They defined an informal algorithm to "calculate" the level of cohesion: *the cohesion of a module is defined as the lowest level of cohesion between all pairs of processing elements*. However, the informal definition of the processing element and associative principles makes cohesion a *subjective measure*.

Lakhotia and Nandigam formally define processing elements and provide an algorithm capable of calculating the level of cohesion [Lak97]. They take output variables as processing elements and examine control and data flow in a variable dependence graph (a graph where nodes are representing identifiers and arcs are standing for dependencies) representing the program. The level of cohesion is then defined by logical rules involving the existence or non-existence of control and data dependencies (see Tab. 6.3) between output variables. The algorithm tests for all levels of cohesion (according to Stevens et.al, beginning with the highest – sequential cohesion) and takes the minimum of the level of cohesion that is detected between these variables.

Lakhotia et.al demonstrate that their algorithm is able to implement the informal algorithm of Stevens et.al, except for temporal cohesion.

According to the principle of structured design, a system is decomposed into several parts, and relationships between them are identified. In addition to the notion of cohesion, Steven, Myers and Constantine suggest another structural design criteria: that of coupling. Coupling is a measure for the strength of intermodule connections. Thus, coupling is a dual measure of cohesion.

Van Vliet names five types of coupling [vV93]. From highest to lowest there are: *content coupling* (where one component directly affects the order of events of another component by changing another component's data or by passing control from one

Cohesion C_i	Rules $rule_i : Var \times Var \rightarrow Bool$
Coincidental	$rule_1(x,y) = \neg\ (\forall_{i \in 2..5} \bullet$ $rule_i(x,y))$
Logical	$rule_2(x,y) = \exists\, z;\ n;\ k \forall\, l \bullet$ $z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,\neg\, k)} y \wedge$ $\neg\ (z \rightarrow_{c(n,l)} x \wedge z \rightarrow_{c(n,l)} y)$
Procedural	$rule_3(x,y) = \exists\, z;\ n;\ k \bullet$ $z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,k)} y$
Communicational	$rule_4(x,y) = \exists\, z \forall\, n;\ l;\ k \bullet$ $\neg\ (z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,\neg\, k)} y) \wedge$ $\neg\ (z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,k)} y) \wedge$ $((z \rightarrow x \wedge z \rightarrow y) \vee (x \rightarrow z \wedge y \rightarrow z))$
Sequential	$rule_5(x,y) = x \rightarrow y \vee y \rightarrow x$
Functional	$\exists_1\, x$ (there exists only one output variable)
Undefined	$\neg\ \exists\, x$ (there are no output variables)

Tab. 6.3: Summary of levels of cohesion between processing elements (out of [Lak97]). x and y denote output variables, $x \rightarrow_{c(n,k)} y$ denotes control dependency between x and y with regard to a statement n that is evaluating a value k (which is either *true* or *false*). $x \rightarrow y$ denotes either control or data dependency between x and y.

component to the middle[11] of another), *common coupling* (where two components share global data), control coupling (where one component directs the execution of another component by passing control-data), *stamp coupling* (when whole data structures are passed between components) and *data coupling* (when simple data is passed from one component to the other).

Generally speaking, it is widely presumed that the more distinctive the structure of a system is, the smaller renders the complexity. This means that even during the development of a system should cohesion already be as high as possible whereas coupling should be as low as possible.

However, up to now there are only informal approaches that detect types of coupling (e.g. between classes in object oriented programming [LJK+01]), and there is no approach *measuring* the level of coupling between components of a system. Coupling is, like cohesion, a qualitative measure.

[11] According to van Vliet a "jump to the middle" means that control is passed from one component to some point within another component.

6.2 Complexity of Specifications

The previous section named several approaches that are used to determine the complexity of a system, be it a program or a design document. Metrics for specifications are not that wide-spread. This section presents existing specification based metrics and compares them to the above mentioned approaches. It discusses their limitation and suggests to use the ASRN as a basis for measuring specification's complexity.

A number of studies (see also Chap. 2.1) show that the use of formal specifications provides quite a lot of benefits. Finney and Fenton [FF96] claim that the use of a formal specification in the SW-development process "leads to code that has a factor of 2.5 times fewer problems[12] than projects not using formal methods". The benefit of the use of formal methods is uncontested, but Finney and Fenton also showed that there are some weaknesses in the study due to the inappropriate use of different metrics[13]. So what kinds of measures do exist? And which metrics are applicable? The subsequent three sections present an overview.

6.2.1 Quantity/Size-based Specification Metrics

Quantity/Size based metrics are simple to calculate automatically. It is no wonder that the majority of specifications' metrics so far used in projects belongs to this class. The following approaches are discussed in the sequel of this section: lines of specification code [SND87], item count [VLK98], fine/large granularity metrics [NLBN00], module count [SND87] and the α-metric [KPHR99].

i. The above mentioned CICS/ESA study uses two factors describing the Z specification: *Lines of Specification Code* and the *time* expended for writing the specification [CNS91]. The same holds for the studies presented in Chap. 2.1. Counting lines is wide-spread, but, again, it is not clearly defined what should be considered as a line in a specification. However, when thoroughly defined the measure of LOC has its benefits. Studies conducted by Samson et. al [SND87] show that the correlation between LOC of a specification and the LOC of its implementation is very high (greater than 0.96).

Lines of (Specification) Code describes the size of a specification, assuming that the more lines there are in the text the more information has to be captured in mind. However, this is not very precise and seems to be less appropriate. In contrast to

[12] The paper of Finney and Fenton refers to a formal-methods-project successfully conducted by the Oxford University's Programming Research Group together with IBM: the CICS/ESA project at the early 1990s.

[13] The most important problem in the CICS/ESA documentation is that there is no complete measure of complexity related to Z and non-Z developments.

programs, lines are not *the* dominant semantic bearing units. Chap. 4 analyzed specifications and identified specification prime objects (primes) as *the* minimal portion of a semantic bearing unit of which specifications are constructed. Thus, it would make sense to *count primes* in order to quantify size-complexity.

ii. Besides looking at the number of lines of specification text, *counting of items* is wide-spread. Vinter et.al [VLK98] count the type and number of logical constructs in Z specifications. Their study indicates that these measures correlate with the complexity of the specification; however, up to now a quantitative assessment of the approach is missing.

iii. Nogueira et.al [NLBN00] suggest to use two complexity measures: the *Fine Granularity Complexity Metrics (FGC)* and the *Large Granularity Complexity Metrics (LGC)*. The FGC expresses the complexity of each operator[14] in the system and is calculated by counting input ($fan-in$) and output ($fan-out$) data related to the operator. The LGC expresses the complexity of the whole system and is based on the number of operators (O), total number of input and output data (D) and the number of types (T) to be found in the specification.

$$FGC \;=\; fan-in \;+\; fan-out \qquad\qquad LGC \;=\; O \;+\; D \;+\; T$$

iv. Samson, Nevill and Dugard [SND87] suggest three measures for specifications. According to their approach a specification consists of a set of modules; each of which defines a set of operations. The following measures are determined: number of equations per operation ($NEQOP$), number of equations per module ($NEQMOD$) and number of operations per module ($NOPS$).

Samson et.al have been inspired by the observation that the number of equations required to define an operator is frequently equal to the cyclomatic complexity of code based on the specification [SND87]. Their basic idea is to apply an approximation to the cyclomatic complexity onto specifications. According to the idea of McCabe, a "1" is added to the complexity each time a logical connective occurs (as this can be considered as an additional path in the control-flow graph). In the approach of Samson et.al no flow-graph is calculated, but the number of predicates is counted. Every predicate adds "1" to the complexity. A small case study shows that there is a correlation between the cyclomatic complexity and the LOC of the implementation[15].

[14] In their terminology an operator is a component defining a specific operation.

[15] The results are based on a relatively small case-study using HOPE as a specification language and Modula-2 for implementation. The correlation between $NEQOP$ and $V(G)OP$ is 0.92, and between $NEQOP$ and LOC of the implementation it is 0.705. The correlation between $NOPS$ and LOC is 0.966, and between $NOPS$ and $V(G)MOD$ it is 0.955. $V(G)OP$ is the cyclomatic complex-

Counting the number of specific operators or input/output variables is another approach where items (not necessarily primes) are counted. In contrast to LOC counts, these items represent units with clearly defined semantic complexity, or at least with comparable semantic complexity. This means that it is possible to say that some items are more complex than others. It also implies that some kind of ordering, some kind of quantification, is possible. For that reason approaches like FGC/LGC and $NEQOP, NEQMOD, NOPS$ are appropriate as complexity measures.

v. Kokol et.al did not only use the **α-metric** for programs but also applied their approach to specifications [KPHR99]. In a case study they tested the metric on a set of specifications languages specifying the popular steam boiler problem [ABL96]. However, the study mainly demonstrates that the α-metric is different for different specification languages. This is not surprising as functionally equal programs written in different programming languages also have different values when calculating complexity measures. The study also demonstrates that the complexity of a specification and the complexity of the generated implementation differ considerably between different specification languages. In general the α-metric, when applied to specifications, at large tends to be lower than the α-metric applied to the corresponding implementation.

   Kokol et. al. omit further discussions and reflections. Thus, up to now the α-metric has no impact on current industrial practice.

Although the α-metric is an outsider it has successfully been applied to specification. However, the entropy of the text is quantified and surely contributes to the complexity of a specification. Kokol admits that the calculation is not simple and for very long texts it is almost impossible to calculate the true entropy according to Shannon's information theory.

Remains one important question: Are size/bulk-based measures enough to quantify a specification's complexity? The answer is quite simple: it is not only size that counts:

1. It is questionable whether specifications of the same size (e.g. LOC) are always of the same complexity.

2. It is also *logical complexity* that counts. The total complexity of a component is not the sum of the complexity of its sub-components. Different logical relationships between items of a component add quite a lot of information, information that has to be considered somehow.

ity of the implemented operation, and $V(G)MOD$ the cyclomatic complexity of the implemented module.

When arguing about the complexity of a specification it seems to be valuable to extend the set of measures. The subsequent section presents some structure-based specification measures.

6.2.2 Structure-based Specification Measures

The application of structure-based metrics is impeded for the following three related reasons:

1. The notion of control- or data-flow is not necessarily a dominant principle of a specification language.
2. As the notion of control is not a dominant principle, formal specifications often do not contain explicit control structures.
3. It is hard to detect control and data dependencies, and thus it is hard to generate a control-flow or data-flow representation of the specification.

Without modifications structure based metrics are not applicable to formal specifications. Due to the simplicity of size-based measures and the problems determining control-flow, it is no wonder that size-based measures are preferred to structure-based measures. However, size-based approaches have been inspired by structure based approaches:

1. The basic idea behind the approach of Samson et.al [SND87] (*NEQOP*, *NOP*, *NEQMOD*) is *cyclomatic complexity*, and thus a structure based metric. They argue that the calculated number for the complexity $v(G)$ equals the number of equations in the specification. As a consequence, this approximately equals the number of "decision statements" in a program. However, their approach does not really take the flow of control into consideration.
2. The approach of Nogueira et.al [NLBN00] (FGC, LGC) makes use of *data-flow considerations*. Again, the approach is not directly based on examining data-flow dependencies. For reasons of simplicity they stick to counting the number of inputs and outputs.

Dependencies are "hard to detect" (argument 3 above). This is in so far true that they are not explicitly visible in the specification source. There are no "*if* ... *then* ... *else*" or "*while* ... *do*" constructs indicating flow of control. However, Chaps. 4 and 5 demonstrated that it is possible to deduce dependencies from specifications. It has also been shown that control- and data-dependencies can be detected quite easily by using an augmented net (ASRN) representing the specification.

With the possibility of determining control and data dependencies it is for the first time possible to consider them directly when calculating specification metrics. It seems reasonable to extend the set of specification metrics by structure-based metrics. A look at existing measures reveals what is possible:

1. Cyclomatic complexity has been introduced in order to measure computational complexity. Every (binary) decision adds further complexity to the system and is thus correlated to independent paths in a control-flow graph. In specifications there is often no extensive flow of control, but there are control dependencies between primes of the specification. When the ASRN is extended by arcs representing control dependencies, then these arcs represent parts of the control-flow of the specification. In most cases (as there is not necessarily a complex flow of control) there will be a set of unconnected arcs representing control and data dependency in the net. Analogous to programs, every arc represents a decision statement. The total number of control-dependency arcs can thus be regarded as an increment of the cyclomatic complexity. For that reason it is suggested to define the cyclomatic complexity of specifications based on control-dependency information deduced from the ASRN.

2. The knot count metric is a measure for the clarity of a program and it is based on control dependencies. The more complicated the flow of control (the more intersections there are), the more complicated it is to read the program. Knots in the flow of control happen when the programming language permits control to be passed across paths (using goto-statements). This is not possible in declarative specification languages. Thus, the knot-count metric is not applicable.

3. The metric of Henry and Kafura is based on the amount of information that is shared and used by a procedure of the program. Data flow within (or better across) the whole program is not considered. However, it is not possible to apply their approach to specifications by just using the ASRN. The ASRN differs between identifiers declared for use in an SRN block, but does not separate input from output identifiers. It would, of course, be possible to extend the ASRN.

4. The $DU(G)$ approach is also concerned with the flow of data, and real data-flow dependencies across the whole program are considered. The $DU(G)$ count reflects *all* data dependencies that are to be identified in the program. With the ASRN this approach can easily be related to specifications. The $DU(G)$ count equals the number of data dependencies in the specification.

To summarize, there is no approach dealing directly with the structural complexity of specifications. However, with the introduction of specification dependen-

Description	Calculation
Cohesion of a module	$Cohesion(\sigma) = \frac{no.\ read\ or\ write\ entries\ for\ module\ \sigma}{total\ no.\ of\ entires\ in\ \tau}$
Coupling between modules	$Coupling(\sigma_1, \sigma_2) = \exists\ shared\ state\ variable\ in\ \Sigma$
Coupling of specification	$Coupling(\Sigma) = \frac{no.\ of\ coupled\ modules}{m!}$
Closeness between two modules	$Closeness(\sigma_1, \sigma_2) = \frac{\sharp(\sigma_1.v\ \cap\ \sigma_2.v)}{\sharp(\sigma_1.v\ \cup\ \sigma_2.v)}$

Tab. 6.4: Quantitative measures for coupling, cohesion and closeness of Z specification modules (out of [CDHW93]). They are based on entries in a cross-reference table τ which summarizes the references to state variables v for each module in the specification Σ. Here σ denotes one schema (module) in the specification. m corresponds to the total number of modules in the specification, and $\sigma_i.v$ is the set of variables accessed by schema σ_i.

cies (and the use of the ASRN) it gets possible to adopt approaches presented in Chap. 6.1.2. It is suggested to make use of the ASRN and to relate two structural complexity measures to the field of specifications: the cyclomatic complexity metrics and the DU(G) count.

6.2.3 Semantic-based Specification Measures

Semantic-based metrics have already been applied to specifications, but there is no approach determining the level of cohesion of components of a specification. The same is true for the type of coupling. However, Carrington, Duke, Hayes and Welsh [CDHW93] define a quantitative measure for *functional* (and partly communicational) *cohesion* and a quantitative measure for *communication coupling* of modules in a specification. Their approach is based on counting both the total number of state variables and the number of state variables that are jointly used by different modules (see Tab. 6.4).

The idea is based on the assumption that a cohesive module is responsible for a single part of functionality of the system. One way to interpret this is to look at how modules operate on the state space. If a subset of operations only requires a subset of the state to define their effect, then one can argue that the whole module is not cohesive and should be split. On the other hand, if modules refer to the same set of state variables, they are said to be coupled.

For a Z specification Σ consisting of m schemata, Carrington et.al build a reference table τ. Rows represent the m schema operations σ and columns represent state variables v. In τ there is an entry (r or w) at position (v, σ) if there is a reference (read or write) to variable v in σ. Taking the table as a basis, they define the notion of *cohesion* of a schema σ, the notion of *coupling* of the overall specification and the notion of *closeness* between two operations (see Tab. 6.4) by simply counting the occurrences of entries in the reference table.

Measures mainly based on	Some representatives for programs/documents	Representatives for specifications
Size/Quantity	LOC [Jon78]	Specification LOC
		Number of Operators †) [SND87]
		FGC/LGC ‡) [NLBN00]
	Halstead's Metrics [Hal77]	–
	α-metric [KPB98]	α-metric [KPHR99]
Structure	Cycl. complexity $v(G)$ [McC76]	–†)
	Knot-count [WHH79]	–
	Information-flow [HK81]	–‡)
	$DU(G)$ [Tai84]	–
Semantic	Slice Profiles [OT89]	Cross-ref. table [CDHW93]
	Rule-bases approach [Lak97]	–

Tab. 6.5: Popular representatives for approaches that are (among other things) used for measuring the complexity of programs and specifications. Approaches for object-oriented programs and (cost-) estimation models have been omitted as there are no comparatives for formal methods. †) The approach is based on the idea of control dependencies and related to $v(G)$, however, only items (equations) are counted. ‡) FCG/LGC considers data-flow in the specification, however, only items (input/output data) are counted.

Chap. 6.1.2 presented another semantic-based approach: the rule-based approach of Lakhotia. However, it cannot fully be transformed into specifications represented by an ASRN. The classification is based on logical expressions that evaluate to a boolean value. That would not be a problem so far, as a pre-condition prime in Z either evaluates to true or false. However, to detect logical or communicational cohesion, control dependency between primes is not enough. The approach requires differentiation between success or failure of the application of the prime. As an example consider the first part of the rule for logical cohesion:

$$rule_2(x, y) = \exists\, z;\ n;\ k \,\forall\, l \bullet z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,\neg\, k)} y \wedge \ldots$$

There is logical dependency between identifier x and y at primes p_x and p_y, if these primes are control dependent on a prime p_z. The logical rules prescribes that p_x is control dependent on p_z and p_x is only executed when p_z evaluates to k (either true or false). It also prescribes that p_y is control dependent on p_z and that p_y is only executed when p_z evaluates to $\neg\, k$. This cannot be detected by using the ASRN as the semantics of the involved operations (k or $\neg\, k$) which is encapsulated by the prime vertices.

Up to now a detailed description of program and specification metrics have been provided. As can bee seen in Tab. 6.5, there are quite a lot of measures that are used

to calculate the complexity of a program, but there are only a few that are used for specifications. It is remarkable that there are no structure-based measures. Here, the ASRN provides a great opportunity. With the ASRN dependencies become explicit for the first time. The remainder of this chapter describes how complexity measures can be calculated by using the ASRN.

6.3 Complexity Measures based on the ASRN

The previous section demonstrated that most of the approaches quantifying the complexity of specifications go back to popular program metrics. When coping with conceptual complexity it suggests to use the ASRN representation as a basis for the metrics calculation. To be more precise it proposes to extend the set of existing specification measures by:

- **Counting specification primes** in order to quantify the information content of specifications. The measure is called Conceptual Complexity *CC* of the specification.
- Calculating the **cyclomatic complexity of the specification** in order to determine the logical complexity of the specification.
- Determine the **DU(G) metric for specifications**, again, in order to determine the logical complexity of specifications.

The following three sections introduce specification measures based on the ASRN. Finally, Chap. 6.3.4 presents a short example to demonstrate the simplicity of applying these measures to Z specifications.

6.3.1 Conceptual Complexity of Specifications

The first measure is called conceptual complexity *CC*. It is based on the number of primes in a specification. In an ASRN prime vertices represent these semantic units.

However, in the ASRN prime vertices represent prime objects *and* higher-level primes. A higher-level prime $FunctioningDB == Add \lor Delete$ consists of three primes (*FunctioningDB*, *Add*, *Delete*) and one higher-level prime $Add \lor Delete$. In the ASRN prime vertices representing higher-level primes do not declare or use identifiers. Thus, when counting primes in the ASRN, higher-level primes have to be excluded (by neglecting vertices with no declaration, definition, or use of an identifier). The conceptual complexity *CC* is defined as follows:

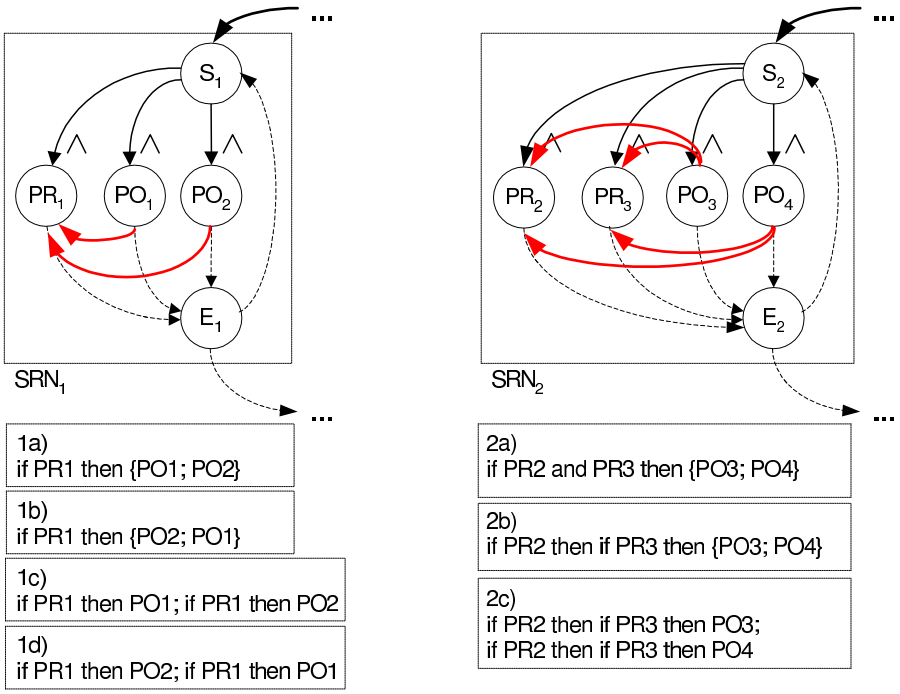


Fig. 6.1: ASRN of a specification and several possible implementations. Cyclomatic complexity is based on control dependencies (in the ASRN drawn as red arcs) going back to decision statements. Even for small specifications and simple logical connectives it is hard to find a suitable relation between specification predicates and program predicates. Statements 1a) to 2c) present different interpretations of two simple SRN blocks which consequently leads to different cyclomatic complexities. ($v(1a) = v(1b) = 2$, $v(1c) = v(1d) = 3$, $v(2a) = 2$ *or* 3, $v(2b) = 3$, and $v(2c) = 5$)

Definition 6.1: Conceptual complexity $CC(\Psi)$ of a specification Ψ. Let $\Upsilon = (N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be an *ASRN* of a syntactically correct Z specification Ψ, and $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$ the corresponding extended SRN.

Then the conceptual complexity $CC(\Psi)$ is the total number of predicate vertices V_{pr} in the eSRN:

$$CC(\Psi) = \sharp\{v : V \mid v \in V_{pr} \wedge (D(v) \cup T(v) \cup T_l(v) \cup C(v) \cup U(v)) \neq \varnothing\}$$

Chap. 6.3.4 presents a short example of how to calculate the conceptual complexity of a specification.

6.3.2 Cyclomatic Complexity of Specifications

The cyclomatic complexity is based on the identification of control dependencies in control flow graphs. McCabe already suggested (as an abbreviation) to look at the number of decision statements that can be identified in the program.

However, relating his approach to specifications is not that easy, as there is no general agreement about how to deal with compound statements[16]. Fig. 6.1 demonstrates that the relation is indeed not trivial. When looking at SRN block SRN_1, there are one pre-condition prime (PR_1) and two post condition primes (PO_1, PO_2). The pre-condition prime represents the decision statement. If it is evaluated to true, the postcondition primes are evaluated, too. This means that there is one decision statement in that particular part of the specification. These pre- and post-conditions can be assigned to $if - then - else$ statements in different ways. Corresponding to program variant 1a) or 1b) a cyclomatic complexity of 2 could be provided.

Another interpretation is that of variant 1c). There are two postcondition primes, two dependencies and thus *two* control-dependency paths to be regarded. In that case a cyclomatic complexity of 3 could be provided.

SRN block SRN_2 demonstrates a case with several pre-condition primes. When assuming a single statement as in variant 2a), the complexity would be 2 or 3, depending on the algorithm that has been chosen (counting logical connectives or not). Variant 2b) treats the logical connective as an extra statement. Finally, variant 2c) does not aggregate the post-conditions, which leads to a maximum number of separate control paths.

Different interpretations lead to values of $v(G)$ between 2..3 for SRN block SRN_1 and between 2..5 for SRN block SRN_2.

A similar situation has been detected for the cyclomatic complexity of programs where rather simple conditions in single statements lead to high complexity measures. In 1977 Myers [Mye77] suggested to extend $v(G)$ to $v'(G) = [l : u]$ where l and u are lower and upper bounds. The lower bound value l equals the number of decision statements plus one. This means that decisions, even when containing sub-expressions, are only counted once. The upper bound value is the number of all control dependencies plus one. In our case this means to count all control arcs in the net.

According to the findings above, first the cyclomatic complexity of a specification is defined by counting all control dependencies in the ASRN. This measure will then be the upper bound value of the extended cyclomatic complexity.

[16] Not all authors add a one to the cyclomatic complexity for every logical connective in the expression of a decision statement.

Definition 6.2: Cyclomatic complexity $v(\Psi)$ of a specification Ψ. Let $\Upsilon = (N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be an *ASRN* of a syntactically correct Z specification Ψ and $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$ the corresponding extended SRN.

Then the cyclomatic complexity $v(\Psi)$ equals 1 plus the total number of control dependencies in the ASRN:

$$v(\Psi) = \sharp\{v_1, v_2 : V \mid v_1 \in V_{pr} \wedge v_2 \in V_{pr} \wedge (v_1 \rightarrow_c v_2)\} + 1$$

The extended cyclomatic complexity is an ordered tuple consisting of a lower and an upper bound value. The value of the upper bound is determined by the cyclomatic complexity as defined above. For the lower bound only dependency arcs which have different initial vertices are counted – thus counting different decision statements. When related to the ASRN, this means counting vertices which represent pre-condition primes that are initial vertices of control dependence arcs.

Definition 6.3: Extended cyclomatic complexity $v'(\Psi)$ of a specification Ψ. Let $\Upsilon = (N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be an *ASRN* of a Z specification Ψ and $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$ the corresponding extended SRN.

The extended cyclomatic complexity $v'(\Psi)$ is an ordered pair (l, u). The upper bound value u equals the cyclomatic complexity of Ψ. The lower bound value l is 1 plus the number of vertices which are terminal vertices of control dependence arcs in the ASRN:

$$l = \sharp\{v_2 : V \mid \exists\, v_1 : V \mid v_1 \in V_{pr} \wedge v_2 \in V_{pr} \wedge (v_1 \rightarrow_c v_2)\} + 1$$

The *extended cyclomatic complexity* v' is defined as an ordered pair consisting of the values of l and $u = v(\Psi)$:

$$v'(\Psi) = (l, u)$$

6.3.3 Definition/Use Count Metric of Specifications

The $DU(G)$ metrics is simple to relate to specifications. Transformed to the ASRN, the measure implies counting the number of data dependencies in the net.

Definition 6.4: DU count metric $(DU(\Psi))$ of a specification Ψ. Let $\Upsilon = (N_{eSRN}, \Sigma_v, T, T_l, C, D, U)$ be an *ASRN* of a syntactically correct Z specification Ψ and $N_{eSRN} = (V, V_{pr}, V_{start}, V_{end}, V_{struct}, V_{comment}, A_c, A_{and}, A_{or}, t)$ the corresponding extended SRN.

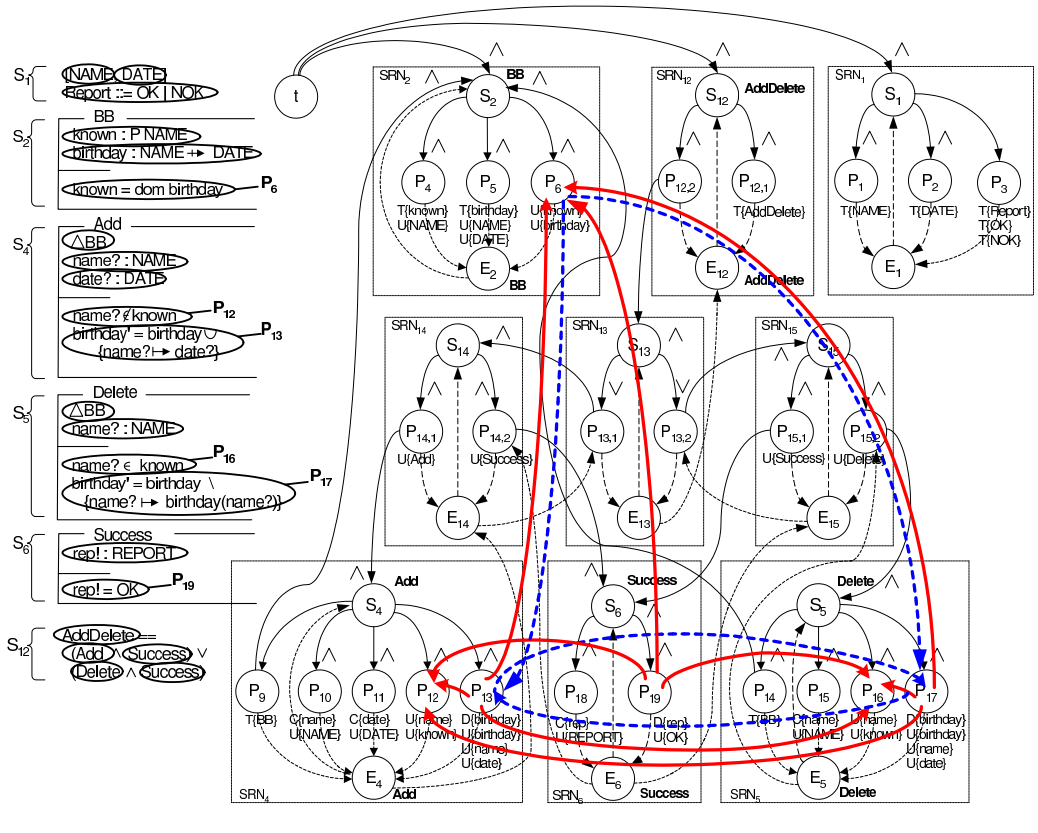


Fig. 6.2: Simplified birthday book specification (on the left) and the corresponding ASRN (on the right) in order to demonstrate the identification of Z measures. Here, the operation schemata *Add*, *Success* and *Delete* are combined using a logical AND and OR operation. Control dependencies are visualized via thick arcs, data dependency is visualized by using thick dashed arcs. Primes are marked by thick circles.

The DU count metrics of a specification equals the total number of data dependencies in the ASRN. It holds:

$$DU(\Psi) = \sharp\{v_1, v_2 : V \mid v_1 \in V_{pr} \wedge v_2 \in V_{pr} \wedge (v_1 \rightarrow_d v_2)\}$$

This completes the transformation of wide-spread program complexity measures to specifications. The next section presents a short example of how these measures are calculated by using the birthday-book specification as an experimental object. Chap. 8 then presents several case studies and demonstrates how these measures are used in order to validate the approach of generating specifications' abstractions.

6.3.4 Calculating Complexity Measures

The previous section relates procedural complexity measures to specifications. The calculation of specification metrics based on the ASRN is quite simple and will be performed step-by-step for the birthday book example in the remainder of this section.

The birthday book specification as presented in Fig. 6.2 consists of 38 lines of specification text written in LaTeX. However, it is not that easy to determine the lines of specification code as it is not clear which lines to count and which lines to skip. So it is quite possible to skip all lines which represent decorations and boxes. When counting all lines (including empty ones and those containing graphical decorations) the BB specification consists of 33 LOC; when neglecting all graphical decoration it consists of 21 LOC which is a difference of more than 36% in size. It is also conceivable that the Z specification is written in its compact notation (horizontal form). The Add schema would then consist of 2 lines of specification code (instead of 6 to 9) and would look like:

$$Add == [\Delta BB;\ name? : NAME;\ date? : DATE \mid \\ name? \notin known;\ birthday' = birthday \cup \{name? \mapsto date?\}$$

For just that reason it is a good idea to stick to counting semantic elements on which the specification is built upon. The conceptual complexity $CC(BB)$ is calculated by counting specification primes. In this example $CC(BB) = 22$ as there are 22 primes in the specification which are assigned to 22 prime vertices in the ASRN. They contain at least a declaration, a definition or make use of an identifier.

For determining the cyclomatic complexity the number of pre-condition primes, post-condition primes and control dependencies is relevant. P_{13}, P_{17}, P_{19} are control dependent on P_6. P_{19} is also control dependent on P_{13} and P_{17}. In fact, there are 9 dependencies to be regarded. The cyclomatic complexity $v(BB)$ is 10.

There are three decision statements (P_6, P_{12}, P_{16}) in the specification and there also three control dependency paths terminating at different prime vertices. For the birthday book the lower bound value of $v'(BB)$ is $l = 4$. Thus, the extended cyclomatic complexity of the birthday book specification is $v'(BB) = (4, 10)$.

The birthday book specification is rather small; nevertheless there are several data-dependencies. *birthday* is modified and used in different primes and primes P_{16}, P_{13}, P_{17} are involved again. Altogether there are 4 data-dependencies in the specification ($P_6 \rightarrow_d P_{13}$, $P_6 \rightarrow_d P_{17}$, $P_{13} \rightarrow_d P_{17}$, $P_{17} \rightarrow_d P_{13}$) leading to a maximum number of 4 (d, u)-tuples. The DU count metric of the birthday book specification is $DU(BB) = 4$.

With the calculation of different complexity metrics it is now possible to grasp the complexity of a specification extensively.

6.4 Summary

This chapter starts with a short introduction to procedural complexity measures. Three classes of complexity measures are identified (size/bulk-based, structure based and semantics-based), and for each class some popular metrics are discussed. Next, the state of the art of complexity measures for specifications is examined and the few existing approaches (Lines of Specification Code, Item count, Fine/Large Granularity Complexity Metrics and the α-Metric) are examined, too. It turns out that size-based measures are commonly used, whereas structure-bases approaches are missing due to the fact that control- and data-dependencies are not explicitly available in the underlying specification.

Nevertheless, size-based measures are not enough for describing complexity. It is not only size that counts. Firstly, specifications of equal size are not necessarily of the same complexity, and secondly, logical complexity cannot be neglected.

It pays off to enrich the set of existing specifications' metrics by structure based metrics. This chapter suggests to use the ASRN as a basis for the calculation as the augmented net makes the necessary dependencies explicit.

The complexity measures mentioned above are examined and discussed in respect to their applicability to specifications and the ASRN. It turns out that most of the existing approaches can be applied to the augmented specification relationship net. Based on the ASRN the following complexity measures are defined:

- Conceptual Complexity $CC(\Psi)$ of a specification Ψ. Here, primes are counted – not the 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 are identified.

With these extensions a wide range of specifications' metrics exists. What is more important is the fact that, based on these quantities, comparisons between different specifications become feasible. This advantage is used in the remainder of this work. Chap. 8 demonstrates that abstractions, as suggested in Chap. 4 and Chap. 5, are really *reducing* the complexity of the underlying specification.

7. COMPREHENSION TOOLKIT PROTOTYPE

Nihil recte sine exemplo decetur.

Columella, res rusticae, XI 1,4

Chap. 5 presented the Augmented Specification Relationship Net. It is used as an alternative representation of a specification and eases the calculation of dependencies. However, for the scope of the previous chapter the transformation of a specification to the ASRN was done by hand. When specifications get larger, however, this step is too laborious.

The ASRN is defined in a language-independent manner, but the transformation rules depend on the specification language at hand. Therefore transformation rules for Z were proposed. Based on these rules, a small prototype has been implemented. The prototype is introduced in more detail in this chapter.

The prototype is designed to be used on Windows and Linux Systems and it is based on a publicly available grammar of Z. It is able to transform Z-specifications (that are type-set in LaTeX) to an ASRN in order to calculate dependencies in the specification. It is also able to transform the ASRN representation back to the specification text.

This chapter is organized as follows: the motivation for the implementation of the prototype and its features are presented in Chap. 7.1. The structure of the prototype and the design decisions are discussed in Chap. 7.2. Finally, limitations and possible improvements are discussed in Chap. 7.3.

7.1 Generation of Abstractions for Z

The generation of an ASRN is straight-forward and Chap. 5.3.3 presented the rules necessary for the transformation of a Z specification (that is written in LaTeX) into the augmented net. Doing the transformation by hand is laborious. The same holds for the identification of dependencies. The birthday-book specification used so far in this work is so small that an automated transformation is not necessary. However, transforming larger specifications is not practicable either.

In order to deal with large specifications, a simple text-based prototype for Z has been implemented. It serves as a basis for the case studies presented in Chap. 8.

The following recurring tasks are handled by the prototype:

1. It generates an ASRN representation from a syntactically correct specification.

2. It calculates dependencies based on the ASRN.

3. It calculates partial specifications, that is Burnstein chunks, static or full static chunks, static specification slices and full static specification slices.

4. It calculates statistics based on the ASRN representation of the specification. This includes the number of primes, the number of control-dependencies (lower and upper-bound values) and the number of data-dependencies.

5. 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.

This subsequent section describes the prototype in more detail. It can be downloaded (in pre-compiled form) from the web-page of the author[1].

7.2 Prototype Description

To ensure portability, the prototype (also 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[2] of Z which has been defined by Breuer and Bowen [BB95]. 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. 7.1).

There are two reasons for splitting the prototype into a front-end (which is based on standard-C and the *preccx* grammar) and a back-end (which is based on Java):

1. The aim is to make the approach (and with it the *SliZe* toolkit) independent from the specification language at hand. When using languages others than Z only the front-end has to be replaced. The back-end remains the same.

[1] Author's web-page: http://www.ifi.uni-klu.ac.at/Andreas.Bollin/private/SliZe. Page last visited: March 2004.

[2] Breuer and Bowen's *preccx* home-page: http://www.afm.lsbu.ac.uk/archive/redo/precc.html. Page last visited: March 2004.

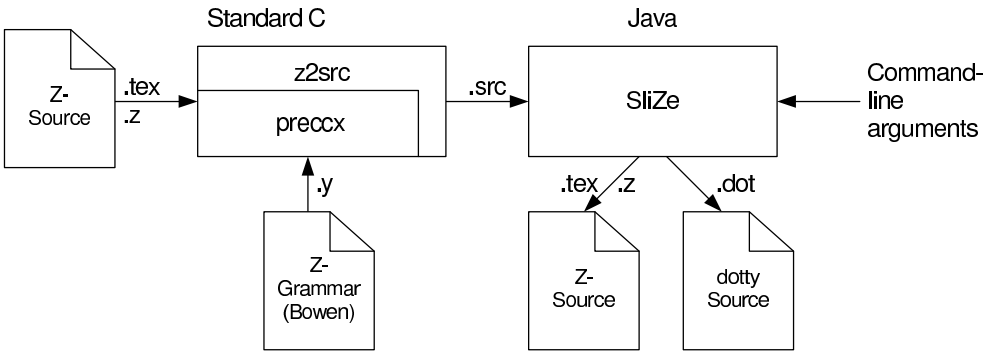


Fig. 7.1: General structure of the prototype. The *preccx* grammar of Z is used to generate an intermediate representation of the specification. This representation serves as an input to the *SliZe* application. The application itself is controlled by command-line arguments and enables both, the generation of partial specifications and the generation of *dotty* representations of the ASRN.

2. The *preccx*-grammar, as defined by Breuer and Bowen, is the only publicly available grammar for Z-specifications which are type-set in LaTeX. It is based on the language definition of Spivey [Spi89b] but has been extended by Breuer and Bowen to cope with LaTeX input. Up to now there is no Z-grammar for Java compilers, and the freely available Java compilers (like JLex[3] and CUP[4]) still have have problems with the ambiguities of the language definition. (These ambiguities are due to mixing Z grammar and LaTeX-grammar. However, they can be resolved rather neatly when using *preccx*[5]).

The prototype is used for the proof of concept and for metrics calculation only. Therefore it is sufficient to control the application via command-line arguments. Displaying the specification by using a graphical interface is appealing, but not without its problems. At the moment the prototype is implemented as text-based application for the following reasons:

1. Mathematical symbols make it hard for Java to display the specification correctly. Remains to display the specification in its LaTeX form of representation. However, a graphical interface would then simply be an overhead.

[3] JLex was developed by Elliot Berk at Princeton University. It is now maintained by C. Scott Ananian. http://www.cs.princeton.edu/ appel/modern/java/JLex. Page last visited: March 2004.

[4] CUP home-page (Scott E. Hudson): http://www.cs.princeton.edu/ appel/modern/java/CUP. Page last visited: March 2004.

[5] There are several reasons why the transformation to another grammar is impeded. The main advantage of *preccx* is that it can handle context-dependent grammars. The reader is referred to [BB95] for more details.

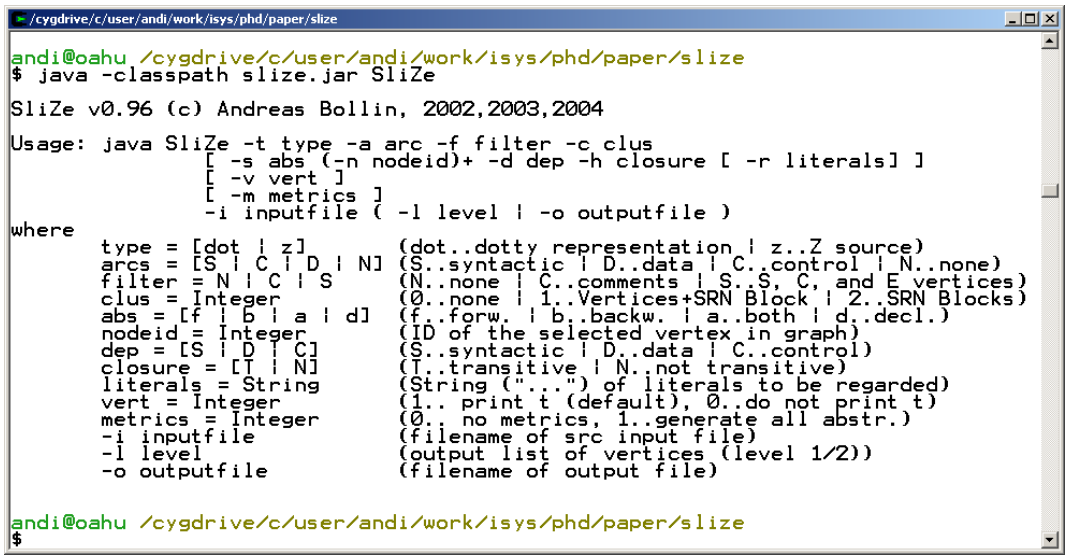


Fig. 7.2: Command-line arguments for the *SliZe* prototype written in Java. It takes an input-file (-i) and optionally writes to an output-file (-o) in either *dotty* (-t dot) or LaTeX (-t z) format. Additionally the LaTeX output can be printed to the screen either by including comments (-l 1) or by omitting comments (-l 2). SRN blocks can be displayed, too. They are either switched off (-c 0), they are displayed and include all vertices of the net (-c 1), or only the SRN blocks are displayed (-c 2). For the generation of partial specifications the type of abstraction (-s), the point(s) of interest (-n), relevant dependencies (-d), the calculation of the closure (-h) and the set of literals (-r) can be provided. Additionally all possible sets of abstractions can be calculated by using the (-m) option.

2. Another solution would have been the integration into an existing environment. Here, the source code is not easy to obtain, and it would only solve the problem of displaying the specification itself. It would still need an extra effort to display the augmented net.

The command-line options (see Fig. 7.2) have proven useful during the generation of partial specifications. By simply switching from the *dotty* mode (-t dot) to the Z-specification source mode (-t z) the prototype is able to generate the LaTeX text representing the partial specification. This specification can be compiled (pretty-printed) to a PDF document and is then visualized in an usual and readable manner. Again, by switching the level of abstraction the net can be displayed in full detail (-t dot -c 0). Otherwise it can be displayed in a more compact manner by just displaying the SRN blocks (-t dot -c 2).

The identification of dependencies is separated from displaying them in the net. The identification is controlled by the (-a)-switch, and the output is filtered by the

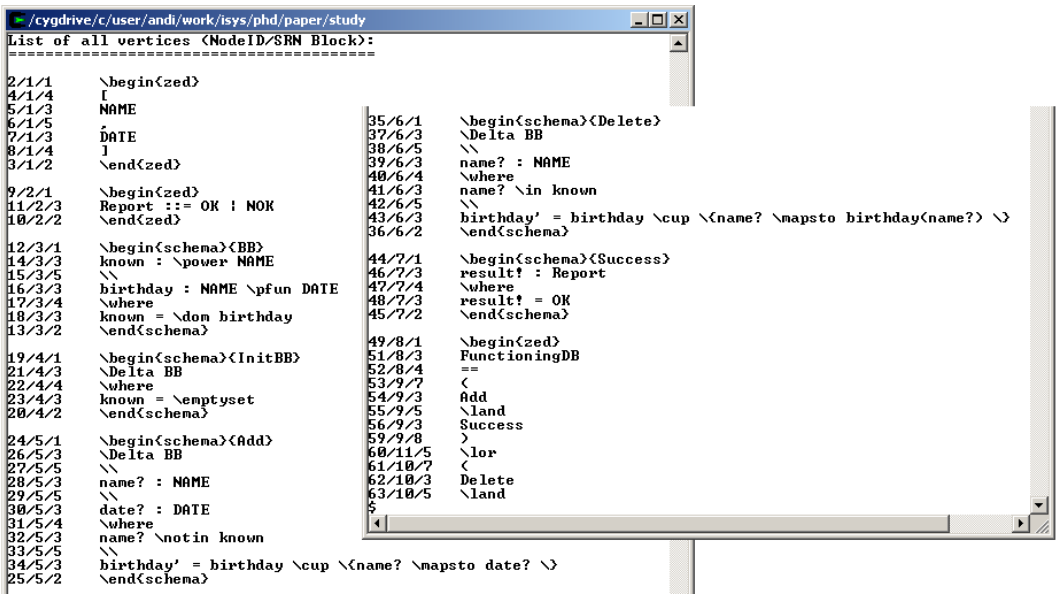


Fig. 7.3: The set of vertices can be displayed in linear form on the screen by using the (-l)-option. This screen-shot of the birthday-book specification presents the output generated by using the option (-l 2) which hides vertices that are comment vertices. Every line represents ASRN information. A line "5/1/3 *NAME*" describes the vertex with vertex-ID "5" which is element of SRN-block "1" and which is a prime vertex ("1" and "2" denote start and end vertices, "3" denotes prime vertices, a value higher than 3 denotes structural vertices). Next to it is the associated LaTeX-text in the specification: the literal "*NAME*".

(-f)-switch. The set of vertices of the ASRN can also be printed to the screen. It is possible to display all vertices represented by their vertex-identifier and the associated LaTeX-text is displayed by using the (-l)-option (see Fig. 7.3). Furthermore, the (-m)-option provides a practical switch in order to automatically generate partial specifications for all possible points of interest (prime vertices) in the specification.

The transformation to the ASRN is based on the rules presented in Chap. 5.3.3. It takes a syntactically correct Z-specification which is type-set in LaTeX input. The output is either a file in *dotty*-format or a Z-specification in LaTeX-format. The subsequent chapter discusses the limits of the application of the prototype and suggests simple improvements.

7.3 Limitations and Improvements

The prototype is needed to calculate dependencies and to generate partial specifications. For reasons of simplicity is makes use of the existing Z grammar and only

implements the ASRN functionality. It has been extensively tested and used with small- and medium-sized Z-specifications. An application to larger specifications is possible but two aspects should be considered:

1. The generation of the ASRN representation can still be done automatically. The result is a *dotty*-source file which serves as an input to the *graphviz/dotty* environment. However, when specifications are getting larger (the ASRN has more than 500 vertices and more than 5000 arcs) *dotty* needs too much time for generating the layout. Rendering the ASRN for the Elevator specification (it consists of 349 vertices and 3668 arcs) takes already two minutes on a Pentium 4 with 1.6 GHz and 512 MB RAM.

2. There is no feedback-loop between the *dotty*-representation of the ASRN and the *SliZe*-prototype. Thus, it is not possible to display the net, then move on to another point of interest and finally restart the calculation by just clicking at this point.

When attempting to improve the prototype for industrial use the two aspects described above have to be dealt with. For a proof of concept it is sufficient to apply the prototype to a specification first and then generate a *dotty* representation of the ASRN. For an industrial use a direct graphical output has to be implemented as it gets necessary to deal with nets consisting of thousands of arcs and several hundreds of vertices. When using *dotty* all the same, one solution is to abstract from the level of vertices and display only SRN blocks. This has already been implemented in the prototype. However, if there are hundreds of SRN blocks, rendering the output takes a lot of computation power again. Thus, a second step might be needed. It is conceivable to abstract from SRN-blocks and display sets of blocks as regions in the net.

Another chance for improvement is the way how the abstraction criteria are provided for the prototype. A criterion consists of three parts: the point of interest in form of a set of prime-vertices, the set of dependencies to be considered and the set of identifiers to be regarded. At the moment the set of prime vertices is provided by a list of vertex-identifiers. These unique identifiers represent the primes in the ASRN. A more comfortable solution would be to click directly at the point(s) of interest in the ASRN and then to start the generation of partial specification.

Finally, the front-end of the prototype is still dependent on the Z-grammar defined for *preccx*. *preccx* and the Z-grammar are not maintained at the moment. To improve and guarantee portability it would be attractive to implement the parser directly in Java. This is possible but, for the reasons mentioned in Chap. 7.2, not without effort.

7.4 Summary

This chapter presented a simple prototype for the generation of partial specifications in Z. In order to simplify porting the prototype to other specification languages it has been split into two parts. The first part, the front-end, has been implemented in standard-C, and it is based on the *preccx*-grammar of Z. With this, a syntactically correct Z-specification can be transformed into an intermediate representation. This representation is taken as an input to the second part of the prototype.

The second part of the prototype is based on Java and implements the ASRN functionality. It generates the augmented net and provides a command-line interface to stipulate the generation of partial specifications. The output is either a *dotty*-file in order to generate an image of the ASRN, or a LaTeX-file in order to pretty-print the Z-specification.

The prototype has some limitations when displaying and interacting with the ASRN. As *dotty* tries to optimize the layout of the net, large specifications lead to very high computation times. The solution is to abstract from the details in the net and to display SRN blocks only. This type of abstraction has already been implemented in the prototype. Another limitation is that the prototype does not provide a graphical interface for user-input. Finally, it still depends on a piece of software that is written in C and impedes portability.

Despite these limitations, the prototype proves to be useful. It simplifies the analysis especially when generating different ASRNs and different partial specifications for the case studies presented in the subsequent chapter. In the context of these studies more then 600 partial specifications have been generated and thousands of dependencies have been detected. And all this within a few minutes.

8. EVALUATION

> We want principles, not only developed-
> the work of the closet-
> but applied,
> which is the work of life.
>
> Horace Mann, Thoughts [1867]

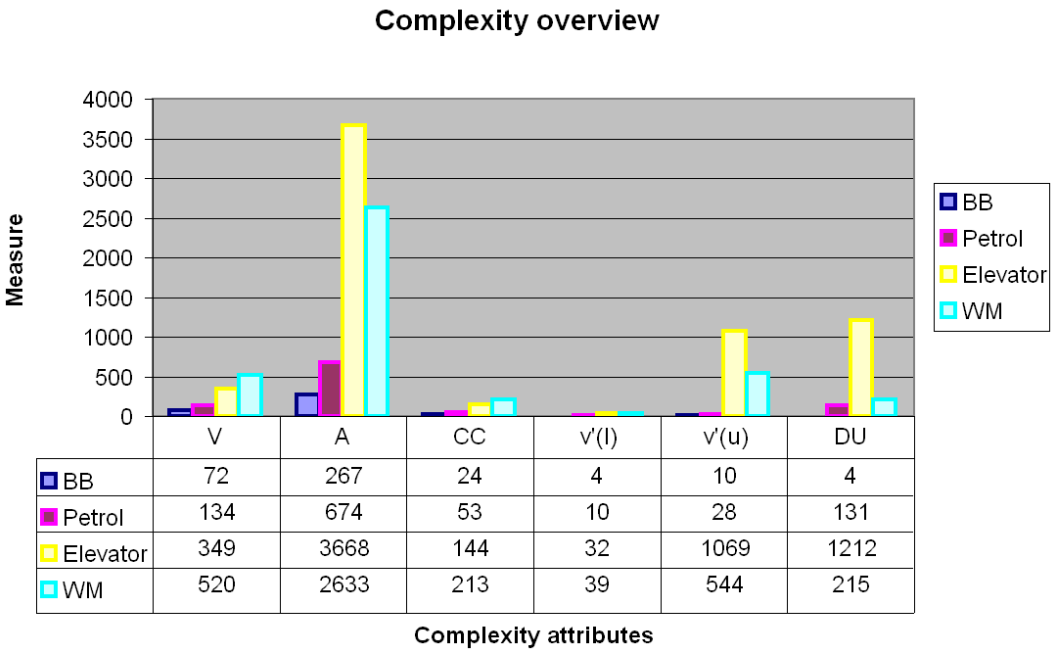
The approach of calculating partial specifications promises to scale down the complexity of specifications, and this chapter provides case studies to underpin this argument. Based on the results of the case studies the question raised in Chap. 2 can be answered: Can a reduction of complexity be expounded, and how can it be measured?

Chap. 6 provided the answer to the second part of this question as it introduced a measure to cope with the complexity of the information content as well as two measures to cope with logical complexity. The objective of this chapter is to validate the concept of partial specifications and to describe the *extent* of the reduction. For this reason more than 600 partial specifications derived from four different specifications (of different and increasing size) have been analyzed.

This chapter is organized as follows: In order to deal with larger specifications the prototype presented in Chap. 7 has been applied to several specifications. The setting for the generation of partial specifications and the treatment are described in Chap. 8.1.1 and Chap. 8.1.2. Then three hypotheses are formulated in Chap. 8.1.3. Based on the results of the treatment these hypotheses are discussed in Chaps. 8.2, 8.3 and 8.4. Finally, this chapter closes with a reflection about the above findings.

8.1 Studies Description

The previous chapters used the birthday-book specification [Spi89b] as a demonstration object so far. It was utilized to demonstrate the transformation to an ASNR, to show how dependencies are identified and to demonstrate the calculation of specification abstractions. Furthermore, it has been used to explain the calculation of complexity metrics.



	V	A	CC	v'(l)	v'(u)	DU
BB	72	267	24	4	10	4
Petrol	134	674	53	10	28	131
Elevator	349	3668	144	32	1069	1212
WM	520	2633	213	39	544	215

Fig. 8.1: 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 $v' = (v'(l), v'(u))$ and the DU count metric.

8.1.1 General Setting

The birthday-book specification is one of the smallest pictorial specifications that are to be found in textbooks introducing Z. Larger specifications are necessary in order to express the effects of partial specifications. 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 [CR94], 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 [Bow96]. The annotated specifications can be found in Chap. C.

Fig. 8.1 summarizes the key attributes of the specifications and visualizes the measures that contribute to their complexity: the total number of vertices (V) in the net, the total number of arcs (A) in the net, 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, the *BB*-specification is the most simple 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.

With respect to the number of vertices, 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 and even the DU metric, the *WM*-specification is less complex.

8.1.2 Experiments

The specifications mentioned in Chap. 8.1.1 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*-specifications are used as experimental objects for the treatment – the application of the generation of partial specifications.

The prototype mentioned in Chap. 7.1 has been used to generate all possible sets of partial specifications for every predicate prime in the specification. These primes are the points of interest[1].

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. The list of points of interest and the mapping between these points and the specification source can be found in App.A.5.

For every point of interest three different types of partial specifications are generated. In fact, the following steps are conducted during the treatment:

1. For *every* prime vertex n, representing a point of interest p in the specification, the full static specification slice $\mathit{FSSlice}(n, \Sigma_v)$ is calculated.

2. For *every* prime vertex n, representing a point of interest p in the specification, the full static specification chunk $\mathit{SChunk}(n, \Sigma_v, \{S, D\})$ is calculated.

3. For *every* prime vertex n, representing a point of interest p in the specification, the full static specification chunk $\mathit{SChunk}(n, \Sigma_v, \{S, C\})$ is calculated.

[1] When applying the prototype to the specifications the (-m 1) option is set.

For all points of interest slices and two types of chunks (one containing control dependencies, one containing data-dependencies) are calculated. The set of identifiers is not restricted when calculating the chunk $SChunk(n, \Sigma_v, \{S, D\})$. Thus, all data dependencies are taken into consideration.

Depending on the specification, this leads to 21 (= 7 points of interest · 3 types of partial specification) possible partial specifications for the *BB*-specification, 63 partial specifications for the *Petrol*-specification, 306 partial specifications for the *Elevator*-specification and 276 partial specifications for the *WM*-specification.

The factors which are assumed to change (and which are called 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 partial specification 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 partial specification) are often considered. E.g. in the *BB*-specification there are 7 points of interest, and for each of these points of interest the full static specification slice is calculated. When only looking at the number of vertices in the resulting ASRNs, every point of interest results in a specific number of vertices in the net ($V_{1..7} = \langle 54, 26, 38, 54, 38, 54, 66 \rangle$). The mean value of these numbers ($\overline{V_{1..7}} = 47,143$) is then considered the average number of vertices when generating a full static specification chunk in respect to the *BB*-specification.

In order to indicate the calculation of the mean in respect to a specific type of partial specification, the function $m()$ is provided. When calculating the mean of mean values, the function $M()$ is provided.

For the specifications at hand, App. A.5 summarizes these mean values and provides statistical information in respect to standard deviation, variance and minimum and maximum values.

The response variables are used to calculate different reduction factors of the specification; e.g., a specification might have been reduced (due to the generation of a specification chunk) by a factor r ($r = 1/k$). As the resulting specification is

smaller than the original one, the corresponding ASRN has also been reduced by a specific factor. The variable $k(V) = V'/V$ describes the reduction of the number of vertices in the net in an inverse manner (V is the number of vertices of the full specification, and V' describes the number of vertices after applying the abstraction criterion to the specification). The dependent variables, called *reduction factors*, are defined as follows:

1. $k(V) = V'/V$ describes the reduction of the number of vertices in the net.
2. $k(A) = A'/A$ describes the reduction of the number of arcs in the net.
3. $k(CC) = CC'/CC$ describes the reduction of the conceptual complexity.
4. $k(v'(l)) = (v'(l))'/v'(l)$ describes the reduction of the lower bound of the extended cyclomatic complexity.
5. $k(v'(u)) = (v'(u))'/v'(u)$ describes the reduction of the upper bound of the extended cyclomatic complexity.
6. $k(DU) = DU'/DU$ describes the reduction of the DU metric.

These reduction factors are used to calculate additional factors describing properties of either the specification or the type of abstraction. The extent of reduction depends on the type of the abstraction. However, to measure the average effect of the approach on attributes of the ASRN and the specification, the mean values of the reduction factors are calculated:

7. $m(k(V))$ describes the mean of reduction of the number of vertices in the net with respect to all types of abstraction.
8. $m(k(A))$ describes the mean of reduction of the number of arcs in the net with respect to all types of abstraction.
9. $m(k(CC))$ describes the mean of reduction of the conceptual complexity with respect to all types of abstraction.
10. $m(k(v'(l)))$ describes the mean of reduction of the lower bound of the extended cyclomatic complexity with respect to all types of abstraction.
11. $m(k(v'(u)))$ describes the mean of reduction of the upper bound of the extended cyclomatic complexity with respect to all types of abstraction.
12. $m(k(DU))$ describes the mean of reduction of the DU metric with respect to all types of abstraction.

Additionally, two factors are calculated in order to indicate the influence of the approach on ASRN- and complexity-attributes of the specification. For that reason the mean value of the reduction of vertices and arcs and the mean value of the reduction of the four complexity attributes $(CC, v'(l), v'(u), DU)$ are provided:

13. $M(k(V,A)) = \overline{m(k(V)) + m(k(A))}$ describes the mean of reduction of ASRN characteristics with respect to all types of abstraction.

14. $M(k(CC, v', DU)) = \overline{m(k(CC)) + m(k(v'(l))) + m(k(v'(u))) + m(k(DU))}$ describes the mean of reduction of complexity attributes with respect to all types of abstraction.

The extent of reduction of vertices and arcs differs. In order to express this ratio another factor is introduced:

15. $f(k) = k(A)/k(V)$ expresses the ratio between the decrease of vertices and the decrease of arcs in the net. A value lower than 1 indicates that arcs are decreased to a greater extent than vertices. $m(f(k))$ expresses the average ratio for all types of abstraction.

The ratio between attributes describing the increasing sizes of specifications and the sizes of calculated partial specifications tell a lot about the efficiency of the approach.

Let $\hat{V}, \hat{A}$ and $\hat{CC}$ be attributes of a specification Ψ_1, and let $m(\hat{V}'), m(\hat{A}')$ and $m(\hat{CC'})$ be the mean values of the attributes of the corresponding partial specifications. Additionally let Ψ_2 be another (larger) specification with attributes V, A and CC. Let $m(V'), m(A')$ and $m(CC')$ be the values of the attributes of the corresponding partial specifications. Then two factors can be calculated:

16. $I()$ describes the increase of the specification in respect to vertices, arcs and conceptual complexity. The increase is calculated as follows: $I(V) = V/\hat{V}$, $I(A) = A/\hat{A}$ and $I(CC) = CC/\hat{CC}$.

17. $Delta()$ 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 was higher than the expected extent of decrease. The factors are calculated as follows: $Delta(V') = m(V')/(m(\hat{V}') \cdot I(V))$, $Delta(A') = m(A')/(m(\hat{A}') \cdot I(A))$ and $Delta(CC') = m(CC')/(m(\hat{CC'}) \cdot I(CC))$

These factors are calculated for all 4 specifications and for the resulting partial specifications. In the subsequent section three hypotheses are presented. Then, based on the dependent and independent variables, these hypotheses are discussed in the remainder of this chapter.

8.1.3 Hypotheses

When generating partial specifications, in most cases one reduces the size of the specification. The resulting partial specification is therefore also less complex. In order to either sustain or refute this assumption, Chap. 6.3 introduced measures that describe conceptual and logical complexity. Observations (that have also been described in [MB03]) show that the reduction factor depends on several aspects of the specification. For that reason, the following (and up to now informally defined) hypotheses are being examined in the remainder of this chapter:

H1 Specification chunks reduce complexity more than specification slices do.

H2 The effect obtainable by slicing and chunking rises with the size of the specification under consideration.

H3 The effect obtainable by slicing and chunking is significant.

The first statement (H1) seems to be obvious. A slice contains all types of dependencies, whereas a chunk ignores some of them. Therefore a chunk cannot be larger than the corresponding slice, moreover, it should be smaller.

The second statement (H2) is based on the observation that partial specifications often represent concepts which the writer of the specification had in mind. Such concepts (e.g. a stack including access functionality) are independent of the size of the specification.

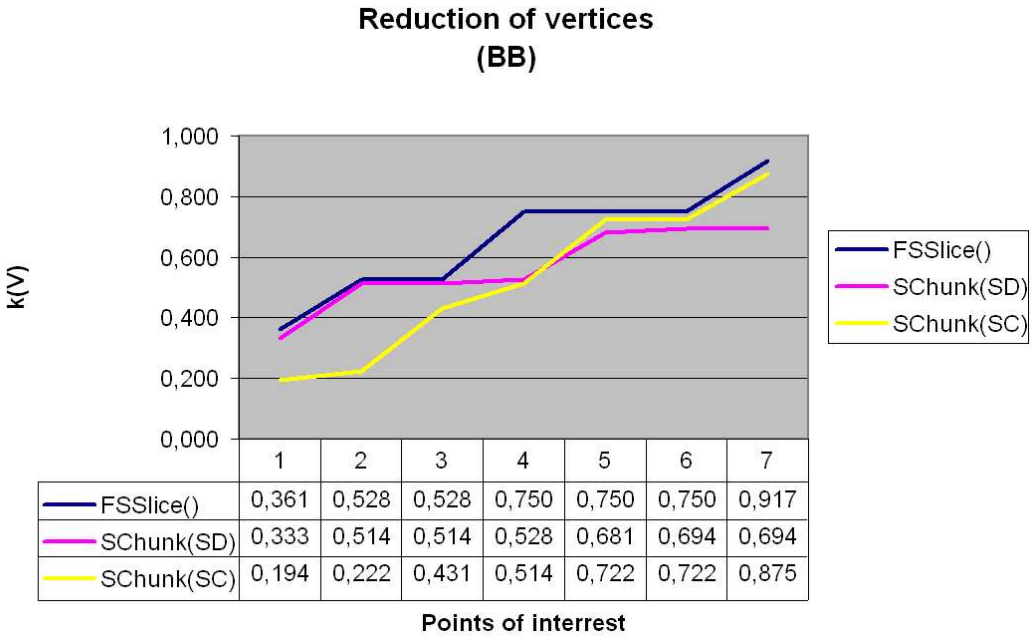
The third statement (H3) is correlated to the second statement as there should be a positive effect when applying the approach to larger specifications. Furthermore, complexity should be reduced efficiently, and Chap. 8.4 provides a definition of efficiency with respect to the approach.

In the subsequent three sections these hypotheses are discussed in more detail. The arguments are based on the evaluation of the data gathered during the case-studies describes in Chap. 8.1.2.

8.2 Extent of Reduction of Complexity

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.

Based on this observation, the following two hypotheses can be stated:



	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

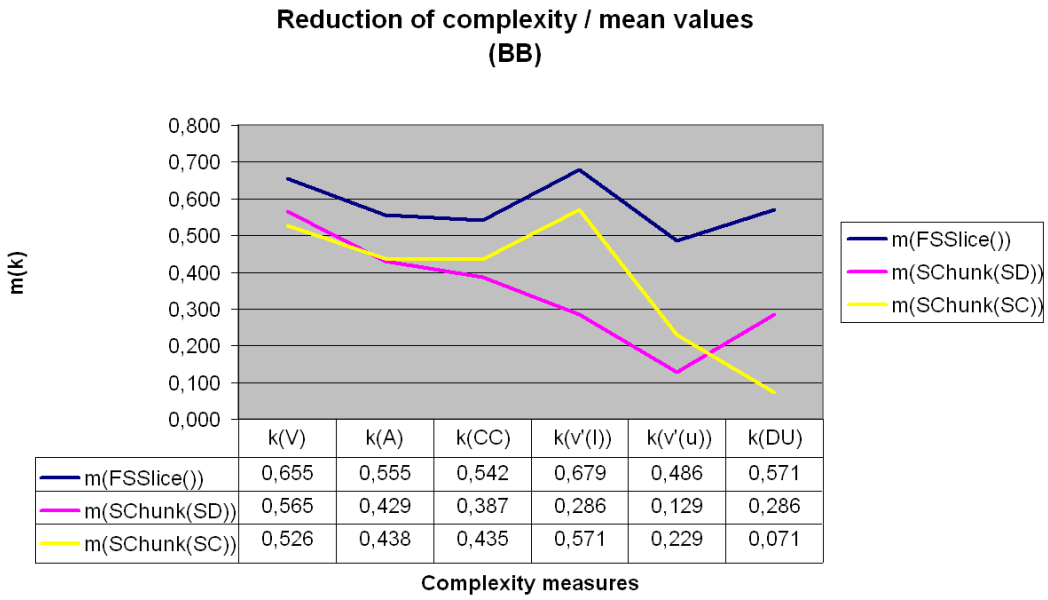
Fig. 8.2: Reduction factor $k(V)$ with respect to the total number of vertices in the ASRN representation of the *BB*-specification. Slices (*FSSlice*()) and chunks (*SChunk*(*SD*), *SChunk*(*SC*)) with respect to the 7 possible points of interest have been calculated. As the reduction factor is defined in an inverse manner, a lower value means a higher reduction.

Null Hypothesis 8.1: There is no difference in mean values of the reduction factors $r = 1/k$ between specification slices and specification chunks.

Alternative Hypothesis 8.1: The mean values $r_1 = 1/k_1$ of the reduction factors of specification chunks are in general higher than the mean values $r_2 = 1/k_2$ of the reduction factors of specification slices. The notion of generally higher mean values r means that for the majority of reduction factors it holds that $r_1 \geq r_2 \Leftrightarrow k_1 \leq k_2$.

Fig. 8.2 visualizes the results of the application of the slicing and chunking approach on the *BB*-specification. For all 7 points of interest slices and chunks have been generated. Then the factors of reduction (with respect to the vertices in the net) have been calculated. It can be observed that there is a reduction of complexity attributes in all cases. This is indicated by values of k that are lower than 1.

Additionally, it can be observed that, with chunks, the values of $k(V)$ are in all cases lower than the values of the corresponding slices. For the first point of interest the extent of reduction is $k(V) = 0.361$ (when generating slices), whereas the extent



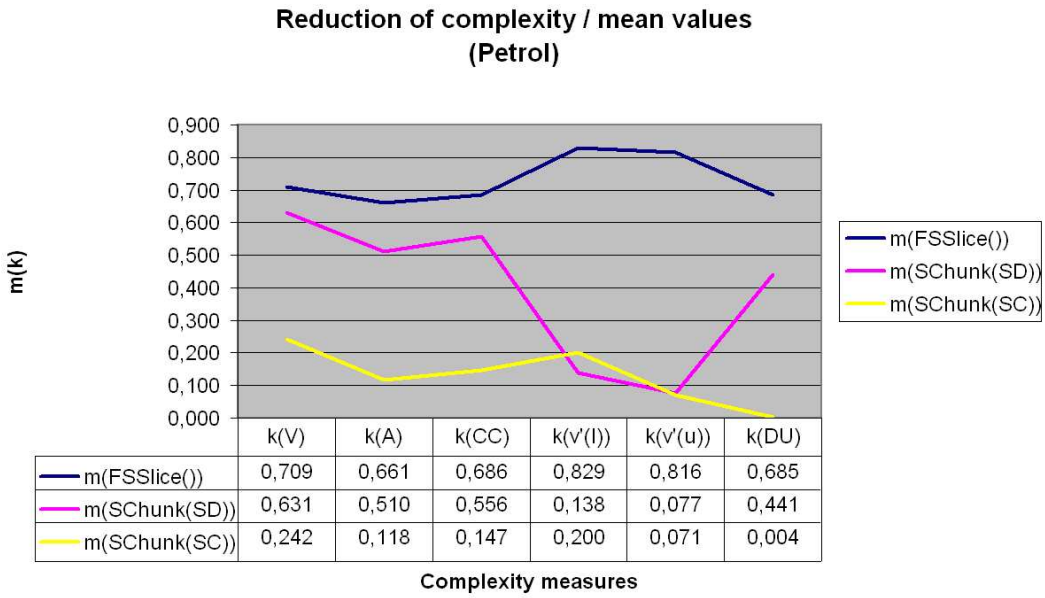
	k(V)	k(A)	k(CC)	k(v'(l))	k(v'(u))	k(DU)
m(FSSlice())	0,655	0,555	0,542	0,679	0,486	0,571
m(SChunk(SD))	0,565	0,429	0,387	0,286	0,129	0,286
m(SChunk(SC))	0,526	0,438	0,435	0,571	0,229	0,071

Fig. 8.3: Mean values of reduction factors with respect to the *BB*-specification.

of reduction is $k(V) = 0,333$ when generating a Burntein chunk. The same holds for the other points of interest. The null hypothesis 8.1 does not hold for all points of interest as $k(V)_{SChunk(SC)} \leq k(V)_{FSSlice}$ and $k(V)_{SChunk(SD)} \leq k(V)_{FSSlice}$.

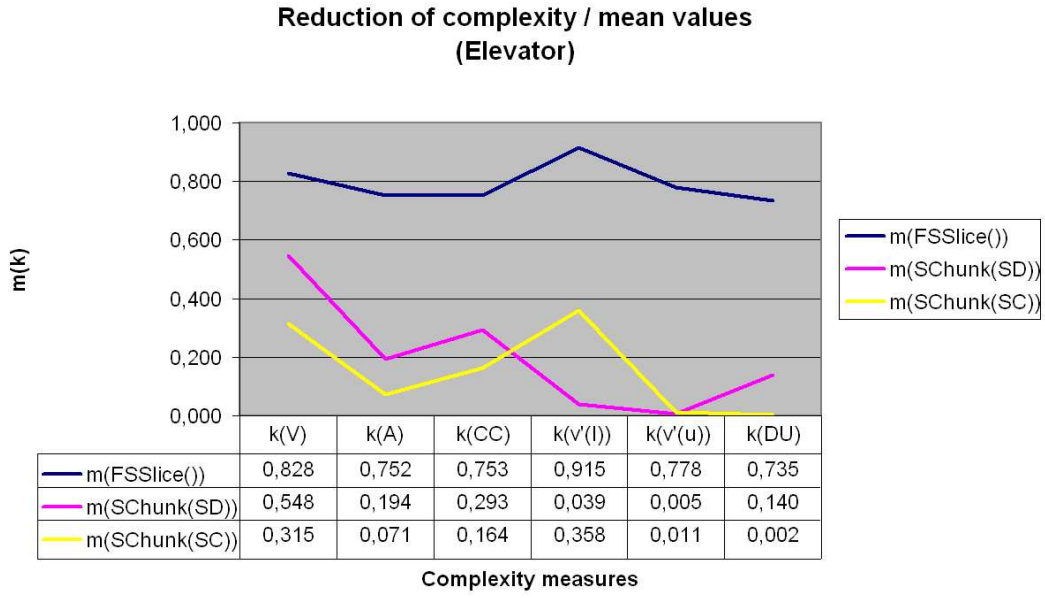
However, this example shows the effect of slices and chunks onto the number of vertices only. It does not show the effect on other complexity attributes. Therefore Fig. 8.3 summarizes the mean values of the reduction factors ($m(k)$) in respect to complexity attributes of the *BB*-specification. In all cases the generation of chunks leads to higher values of the reductions. The same holds for the *Petrol*-specification (Fig. 8.4), the *Elevator*-specification (Fig. 8.5) and the *WM*-specification (Fig. 8.6). The disadvantage of neglecting information becomes an advantage as the focus gets sharper.

When looking at these figures it can be observed that slices do not guarantee much smaller (and less complex) specifications. As the full static specification slice is calculated, all dependent (and indirectly dependent) primes are included in the resulting specification. The mean value of the reduction factor r is higher. For vertices in the *BB* it holds that $m(k_{FSSlice()}) = 0,655$, $m(k_{SChunk(SD)}) = 0,565$ and $m(k_{SChunk(SC)}) = 0,526$. This implies that when generating slices the reduction is about $r = 1,527 (= 1/0,655)$. When generating chunks the reduction is in between $r = 1,770 (= 1/0,565)$, when applying $SChunk(SD)$, and $r = 1.901 (= 1/0,526)$, when applying $SChunk(SC)$ to the specification.



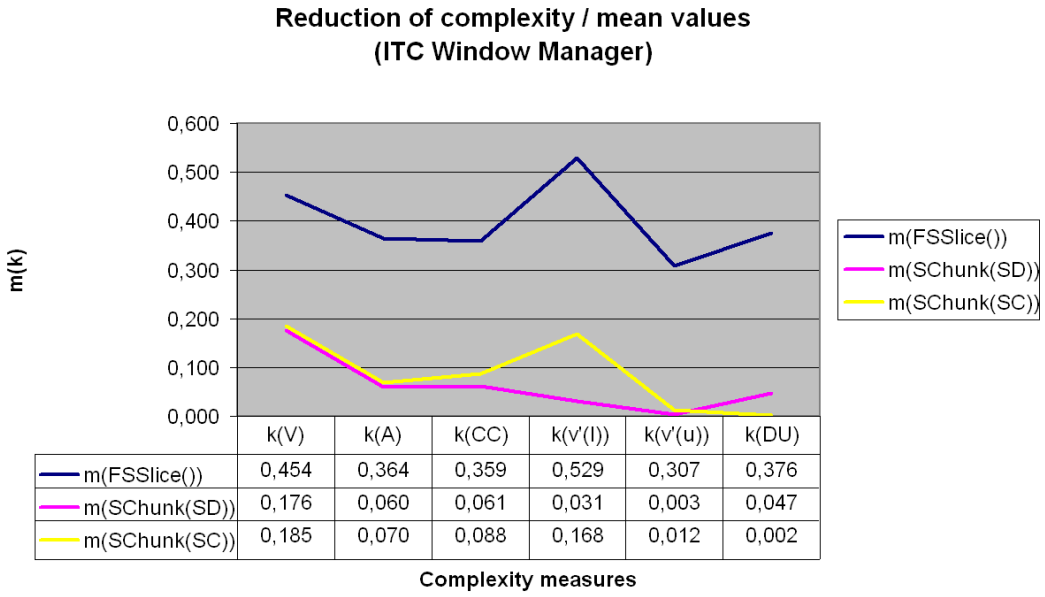
	k(V)	k(A)	k(CC)	k(v'(l))	k(v'(u))	k(DU)
m(FSSlice())	0,709	0,661	0,686	0,829	0,816	0,685
m(SChunk(SD))	0,631	0,510	0,556	0,138	0,077	0,441
m(SChunk(SC))	0,242	0,118	0,147	0,200	0,071	0,004

Fig. 8.4: Mean values of reduction factors with respect to the *Petrol*-specification.



	k(V)	k(A)	k(CC)	k(v'(l))	k(v'(u))	k(DU)
m(FSSlice())	0,828	0,752	0,753	0,915	0,778	0,735
m(SChunk(SD))	0,548	0,194	0,293	0,039	0,005	0,140
m(SChunk(SC))	0,315	0,071	0,164	0,358	0,011	0,002

Fig. 8.5: Mean values of reduction factors with respect to the *Elevator*-specification.



	k(V)	k(A)	k(CC)	k(v'(l))	k(v'(u))	k(DU)
m(FSSlice())	0,454	0,364	0,359	0,529	0,307	0,376
m(SChunk(SD))	0,176	0,060	0,061	0,031	0,003	0,047
m(SChunk(SC))	0,185	0,070	0,088	0,168	0,012	0,002

Fig. 8.6: Mean values of reduction factors with respect to the *WM*-specification.

It is observable that chunks do a great job when it comes to reducing logical complexity. With the *BB*-specification the number of control dependencies is reduced by factors $r = 7,75(= 1/0,129$ and $r = 4,37(= 1/0,229)$; the number of data dependencies is reduced by factors $r = 3,5(= 1/0,286)$ and $r = 14,08 = (1/0,071)$. The effect is higher when applied to larger specifications. With the *WM*-specification the number of control dependencies is reduced by factor $r = 333(= 1/0,003)$ and $r = 83,33(= 1/0,012)$; the number of data dependencies is reduced by factor $r = 21,28(= 1/0,047)$ and $r = 500(= 1/0,002)$.

The results summarized in Figs. 8.3, 8.4, 8.5 and 8.6 also show the difference between chunks in respect to control-dependencies and data-dependencies. The measure DU decreases when calculating the chunk $SChunk(SC)$ as data-dependencies are not considered during the generation of the partial specification. And the measure $v'(u)$ decreases when calculating the chunk $SChunk(SD)$ due to the fact that control-dependencies are not in the focus of that type of partial specification.

Summarizing the above results it can be stated that the null hypothesis 8.1 does not hold. On the other hand, the alternative hypothesis 8.1 is confirmed. In all treatments the mean of the reduction $r = 1/k$ is higher when applying specification chunks. It is noticeable that the extent of reduction (e.g. conceptual complexity CC) increases when looking at larger specifications. In the subsequent section this assumption is discussed in more detail.

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**		

Fig. 8.7: 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, an SD and SC indicate the generation of a specification chunk ($SChunk(SD)$, $SChunk(SC)$).

8.3 Influence of the Specification's Size

The second hypothesis states that 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, with larger specifications the extent of the reduction should increase, too. The following case study investigates the influence of the size on different complexity measures.

Again the *BB*-, *Petrol*-, *Elevator*- and *WM*-specification are taken as experimental objects. 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. Independent and dependent variables are the same as presented in Chap. 8.2. However, to describe the average case the mean values (M) of all reduction factors are of interest. Fig. 8.7 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.

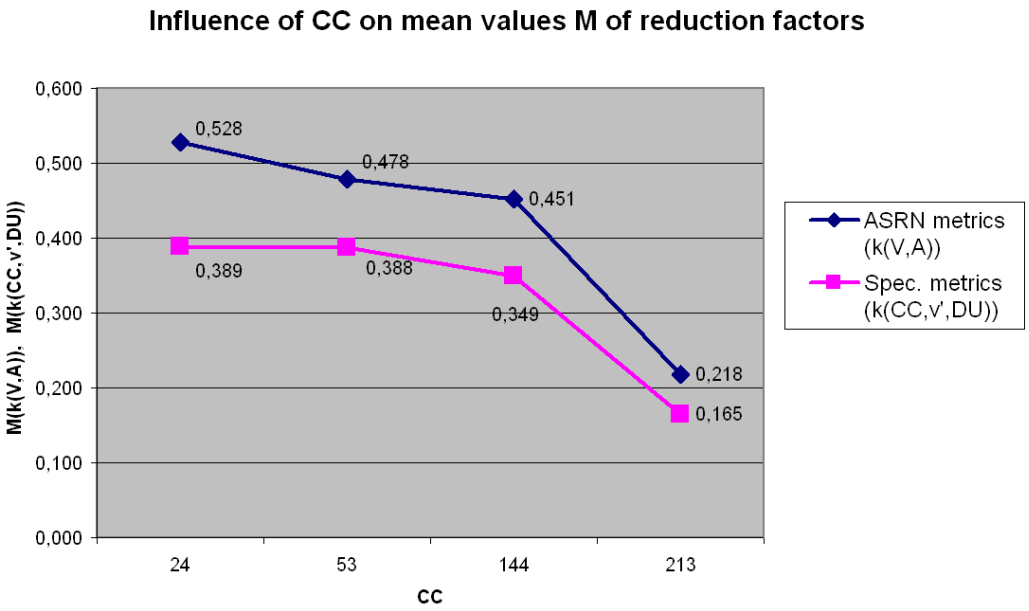


Fig. 8.8: Relationship of the size of the specification (described by CC) and the mean values M of ASRN and specification-related reduction factors. With increasing size of the specification the factors $M(k(V, A))$ and $M(k(CC, v', DU))$ decrease. This means that the extent of reduction increases with larger specifications.

Based on the response variables and reduction factors the following two hypotheses can be stated:

Null Hypothesis 8.2: The mean values of the reduction factors $r = 1/k$ for ASRN and specification attributes $M(k(V, A))$ and $M(k(CC, v', DU))$ stay the same or increase when the treatment is applied to larger specifications (size is described by CC).

Alternative Hypothesis 8.2: The mean values of the reduction factors $r = 1/k$ for ASRN and specification attributes $M(k(V, A))$ and $M(k(CC, v', DU))$ decrease when the treatment is applied to larger specifications (size is described by CC).

Fig. 8.8 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 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 complexity is reduced by a factor of $6,06$ $(= 1/0,165)$.

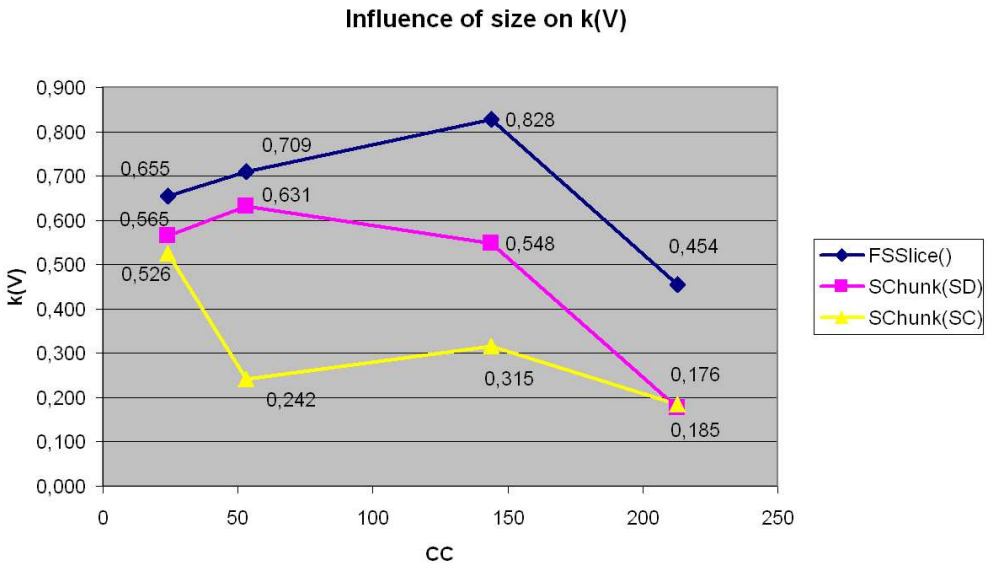


Fig. 8.9: Relationship of the size of the specification (described by CC) and the mean values of k(V). The figure is based on the values presented in Fig. 8.7.

The mean value of the reduction factors r increase with increasing size of CC. Thus, the null hypothesis 8.2 is refuted, and the alternative hypothesis 8.2 is proven true. However, by using mean values two aspects of the treatment are neglected: the influence of the type of the specification at hand and the influence of the type of abstraction. Specifications differ from each other in the number of interrelated primes and existing dependencies. Slices and chunks are different approaches for different problems.

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 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.

Fig. 8.9 visualizes that the effects strongly depend on the type of the applied abstraction (slice or chunk). When looking at the generation of specification slices ($FSSlice()$) factor $k(V)$ first increases relative to the extent of reduction of the smaller specifications. This effect can also be observed in Fig. 8.7 by looking at the lines marked with the abbreviation "(SCD)". Taking the reduction factor $k(V)$ as a measure for the extent of reduction, the values for the four specifications differ considerably.

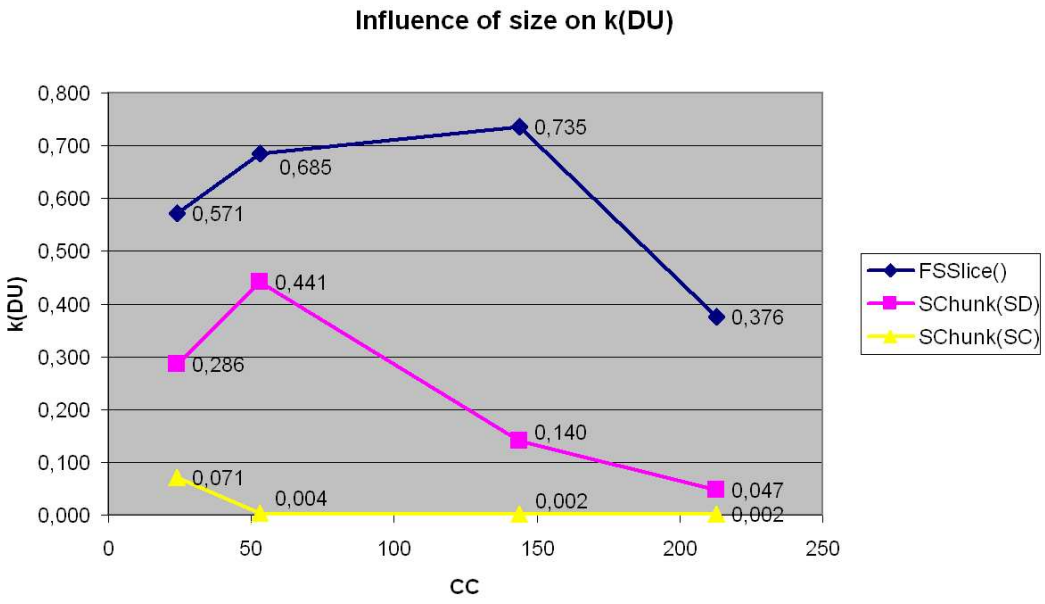


Fig. 8.10: Relationship of the size of the specification (described by CC) and the mean values of $k(DU)$. The figure is based on the values presented in Fig. 8.7.

For the *BB*-specification it holds that $k(V) = 0,655$, for the *Petrol*-specification $k = 0,709$ and for the *Elevator*-specification $k = 0,828$. Only for the *WM*-specification the factor decreases again. The same holds for the number of data dependencies (see Fig. 8.10). The extent of $k(DU)$ increases from $k = 0,571$ (*BB*) up to $k = 0,735$ (*Elevator*) and then decreases to $k = 0,367$ (*WM*). Thus, only in the long range and in the average case the extents of factors $m(k)$ decrease. As Figs. 8.9 and 8.10 illustrate, the decrease is better noticeable when looking at chunks only.

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 (e.g. *FSSlice*()), hypothesis 8.2 does not hold. Taking a closer look at the generated specification slices, the value of $k(V)$ *increases* with increasing size of the specifications.

Fig. 8.9 and Fig. 8.10 demonstrate (for the three types of partial specifications) that the extent of the reduction (of the number of vertices in the net and the number of data dependencies) even decreases during the generation of specification slices. The reason is the influence (the number and type of interrelated dependencies) of the specification at hand. This influence can be observed on complexity measures other than $k(V)$ and $k(DU)$, too. App. A.4 summarizes the values for the other types of complexity measures.

Summarizing the above observations it can be stated that the effect obtained by partial specifications increases with larger specifications. On the other side this statement only holds when dealing with different types of specification abstractions. Especially slicing does not guarantee smaller specifications, and so the hypothesis cannot be confirmed when applying specification slices only. It is still an open question if the approach is effective, and the objective of the subsequent section is to clarify the significance of the approach with respect to reducing complexity factors.

8.4 Efficiency of the Approach

Two aspects are important when applying slicing or chunking techniques to specifications: the benefits should increase with increasing size of the specifications at hand. This means that the generated partial specifications 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.

Definition 8.1: Efficiency of the approach. The approach of generating partial specifications 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).

ad Def. 8.1.a) The first part of the definition of efficiency deals with the *size* of the specification and the resulting partial specification. For this reason it also has to do with the influence of size onto reduction factors (Chap. 8.3). However, for the scope of this chapter 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.

ad Def. 8.1.b) The comparison of conceptual (CC) and logical (v', DU) 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 $f(k)$ (see Chap. 8.1.2). Vertices represent conceptual entities and arcs represent logical dependencies. The basic assumption is that the reduction of the number of vertices leads to a higher reduction of arcs in the net.

Formally, if 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:

$$\begin{aligned} &a = O(3 \cdot n^2) \\ &a' = O(3 \cdot (n/k_v)^2) \qquad\qquad \textit{with } k_v = O(n/(n')) \\ &k_a = a/(a') = O(\tfrac{3 \cdot n^2}{3 \cdot (n/k_v)^2}) = O(k_v^2) \end{aligned}$$

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.

Based on the response variables and dependent variables, the following hypotheses can be stated for the efficiency of the approach. As the definition is split into two parts, hypothesis (H3) is also split into two parts. First, the hypotheses in respect to Def. 8.1.a is stated. (For $M(Delta)$ see Chap. 8.1.2):

Null Hypothesis 8.3: The mean value of the sizes of partial specifications (described by V', A', CC') increases to the same extent as the sizes (based on CC) of the underlying specifications. An increase of CC (of the specification at hand) by a factor I results in an increase of size and complexity (of the partial specification) by a factor $I' \geq I$. It holds that $M(Delta()) \geq 1$.

Alternative Hypothesis 8.3: The mean value of the sizes of partial specifications (described by V', A', CC') increases at 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 $M(Delta()) \leq 1$.

The second part of the definition of efficiency (Def. 8.1.b) deals with the *ratio* between logical and conceptual complexity:

Null Hypothesis 8.4: The reduction of the logical complexity is, at the most, as high as the reduction of the conceptual complexity. Thus, for the generation of all possible abstractions it holds that $M(f(k)) \geq 1$.

Alternative Hypothesis 8.4: The logical complexity is at least reduced to the same extent as the conceptual complexity. Thus, for the generation of all possible abstractions it holds that $M(f(k)) \leq 1$.

Fig. 8.11 summarizes the values for the response and for the dependent variables. Based on these results the null and alternative hypotheses are discussed in the sequel of this section.

Spec.(Deps.)	V	A	CC	V'	A'	CC'	M(Delta)
BB Total	**72**	**267**	**24**				
BB (SCD)				47,143	148,143	13,00	
BB (SD)				40,714	114,571	9,29	
BB (SC)				37,857	117,000	10,43	
Mean				**41,905**	**126,571**	**10,905**	
Increase I	1,00	1,00	1,00				
Approx.				41,905	126,571	10,905	
Delta				**1,000**	**1,000**	**1,000**	**1,000**
Petrol Total	**134**	**674**	**53**				
Petrol (SCD)				95,000	445,429	36,38	
Petrol (SD)				84,619	343,667	29,48	
Petrol (SC)				32,381	79,429	7,81	
Mean				**70,667**	**289,508**	**24,556**	
Increase I	1,86	2,52	2,21				
Approx.				77,989	319,510	24,081	
Delta				**0,906**	**0,906**	**1,020**	**0,944**
Elevator Total	**349**	**3668**	**144**				
Elevator (SCD)				289,059	2759,765	108,36	
Elevator (SD)				191,167	710,216	42,16	
Elevator (SC)				110,000	261,284	23,57	
Mean				**196,742**	**1243,755**	**58,029**	
Increase I	4,85	13,74	6,00				
Approx.				203,122	1738,816	65,429	
Delta				**0,969**	**0,715**	**0,887**	**0,857**
WM Total	**520**	**2633**	**213**				
WM (SCD)				236,087	958,087	76,41	
WM (SD)				91,739	159,283	13,08	
WM (SC)				96,217	183,696	18,71	
Mean				**141,348**	**433,688**	**36,065**	
Increase I	7,22	9,86	8,88				
Approx.				302,646	1248,174	96,780	
Delta				**0,467**	**0,347**	**0,373**	**0,396**

Fig. 8.11: Table summarizing the number of vertices, arcs and the extent of CC before and after the creation of the partial specifications. The table is split into several parts. The first part to the left summarizes measures of the specification at hand (V, A, CC) and provides the extent of the increase I. The third part presents the mean values V', A' and CC' of the sizes of the partial specifications and the *Mean* of these sizes. Additionally, *Approx* represents the numbers of $\tilde{V}'$, $\tilde{A}'$ and $\tilde{CC}'$ which are estimations based on the extent of increase. *Delta* expresses the difference (ratio) between the true and the estimated values. To the right, $M(Delta)$ provides the mean of these differences in respect to V, A and CC.

For Hypothesis 8.3 one has to look at the sizes of the specifications. By looking at the ratio factor *Delta* and the mean of *Delta* for V, A and CC, it is possible to determine whether the increase in size and complexity is the same or not.

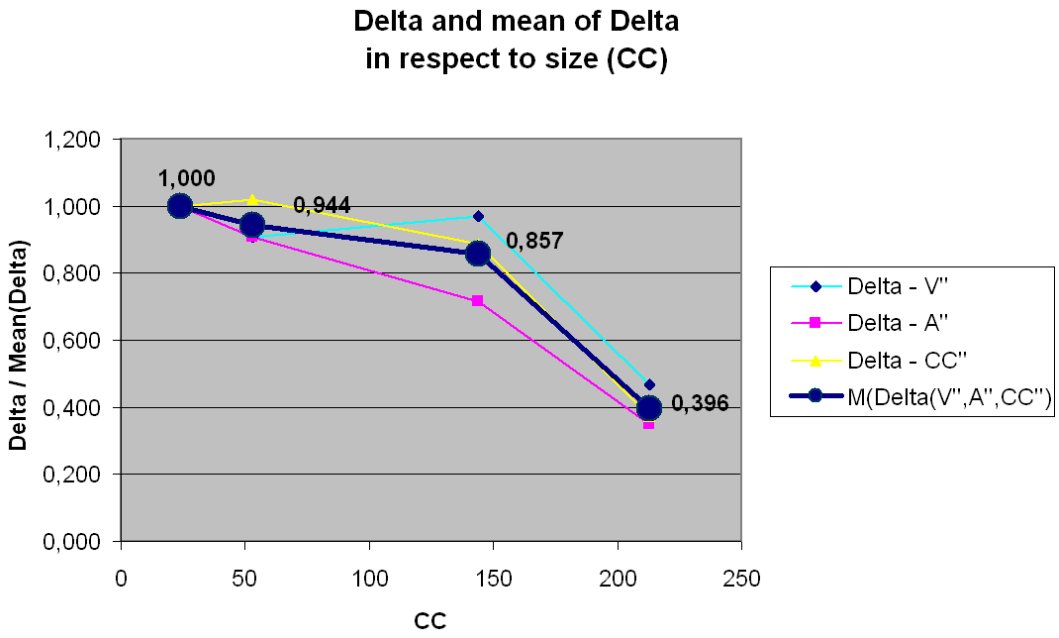


Fig. 8.12: Effect of reduction when increasing the size of the specification (described by CC). If the effect increases at the same ratio as the original specification, then the value of *Delta* should be 1. A value of *Delta* lower than 1 indicates that the approach is more efficient.

A value of 1 (or less than 1) indicates that the increase of complexity measures is equal or less than the increase of the size of the specification. This indicates that the approach is efficient in terms of Def. 8.1.a. The complexity of the partial specifications 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 if the size of the specification increases by a factor of 2, the approach generates partial specifications that also increase by a factor of 2.

As can be seen in Fig. 8.12 (which displays the values of *Delta* in Fig. 8.11) 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 partial specification contains $V' = 70,667$ vertices in the mean, and not $\tilde{V}' = 77,989$ vertices. The approach has been a bit more effective, and 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. For that reason the null hypothesis 8.3 is refuted and the alternative hypothesis 8.3 is confirmed.

Spec.(Deps.)	V	A	k(V)	k(A)	f(k)	CC
BB Total	**72**	**267**				**24**
BB (SCD)	47,143	148,143	0,655	0,555	0,800	13,000
BB (SD)	40,714	114,571	0,565	0,429	0,729	9,286
BB (SC)	37,857	117,000	0,526	0,438	0,814	10,429
Mean	**49,429**	**161,679**	**0,582**	**0,474**	**0,781**	**14,179**
Petrol Total	**134**	**674**				**53**
Petrol (SCD)	95,000	445,429	0,709	0,661	0,835	36,381
Petrol (SD)	84,619	343,667	0,631	0,510	0,730	29,476
Petrol (SC)	32,381	79,429	0,242	0,118	0,479	7,810
Mean	**86,500**	**385,631**	**0,527**	**0,430**	**0,682**	**31,667**
Elevator Total	**349**	**3668**				**144**
Elevator (SCD)	289,059	2759,765	0,828	0,752	0,863	108,363
Elevator (SD)	191,167	710,216	0,548	0,194	0,326	42,157
Elevator (SC)	110,000	261,284	0,315	0,071	0,219	23,569
Mean	**234,806**	**1849,816**	**0,564**	**0,339**	**0,469**	**79,522**
WM Total	**520**	**2633**				**213**
WM (SCD)	236,087	958,087	0,454	0,364	0,638	76,413
WM (SD)	91,739	159,283	0,176	0,060	0,323	13,076
WM (SC)	96,217	183,696	0,185	0,070	0,319	18,707
Mean	**236,011**	**983,516**	**0,272**	**0,165**	**0,427**	**80,299**

Fig. 8.13: Table that summarizes the values for $V, A, k(V)$, $k(A)$ and the factor $f(k)$.

When discussing hypothesis 8.4 one has to look at the relationship between the reduction factors $k(V)$ and $k(A)$. In fact, 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 $r_V = 1/k(V)$. The factor $f(k)$, calculated for the experimental objects, is presented in Fig. 8.13 in more detail.

When looking at the factor $f(k)$ 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 also 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. 8.14 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. For that reason the null hypothesis 8.4 is refuted and the alternative hypothesis 8.4 is confirmed.

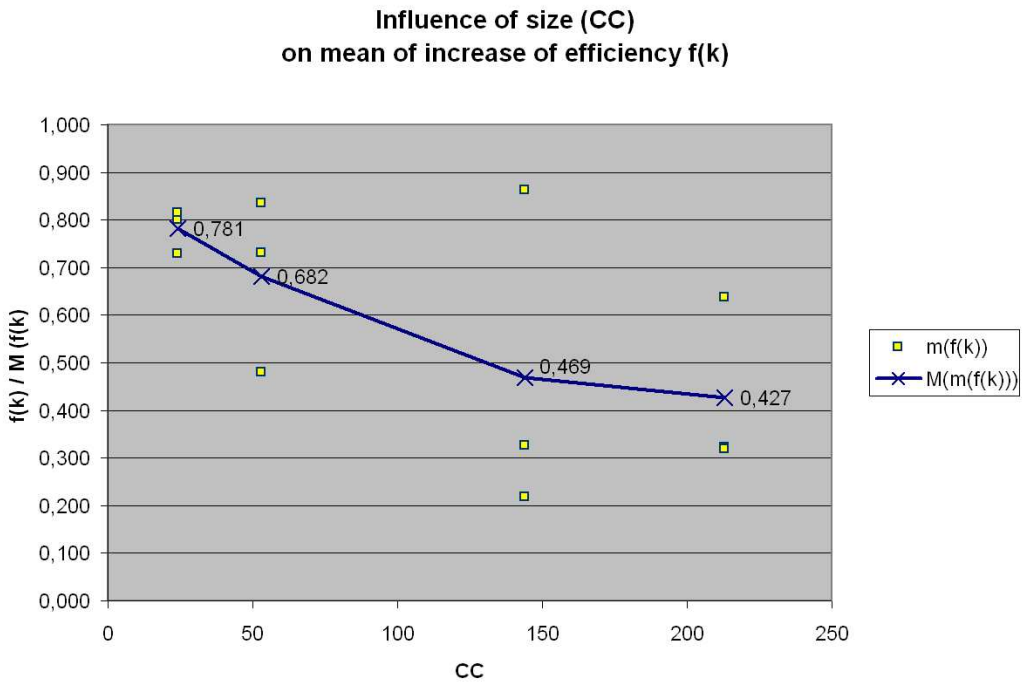


Fig. 8.14: The effect of reduction (described by the mean value of reduction $f(k)$) increases with the size of the specification (described by CC). If the effect increases at the same ratio as the original specification, then the value of $m(f(k))$ should be 1. Additionally, the mean of all reductions ($M(m(fk))$) is presented. For the values of $m(f(k))$ and $M(m(f(k)))$ see Fig. 8.13.

The above figures only present the mean values for the reductions and the factor $f(k)$. A more detailed view of the influence of the size on other measures can be found in then appendix in App. A.4.

8.5 Summary

This chapter discusses the efficiency of the generation of partial specifications. The discussion includes

- the influence of the different types of abstraction criteria,
- the effect of the approach on larger specifications and
- the efficiency of the approach.

Three hypotheses are stated. In order to refute or confirm these hypotheses, four specifications of different (but increasing) size are taken as experimental objects.

The case studies are conducted by using the simple prototype described in Chap. 7.1. Bases on the four experimental objects all possible types of partial specifications for all points of interest have been calculated. The case studies confirm the following hypotheses:

H1 Specification chunks reduce complexity more than specification slices do. 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.

H2 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.

H2 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 partial specification is definitely less than the increase of the size of the specification.

In total more than 600 partial specifications have been examined. 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, and it is likely that the approach still proves useful. The results imply that with larger specifications there is generally a trend toward higher extents of reduction.

This completes the reflection on the efficiency of generating specification slices and chunks. The approach of generating partial specifications has proven to be useful and effective. The last chapter of this work summarizes all the findings.

9. CONCLUSION

Finis coronat opus.

Ovid, Heroides II 85

In this work I have argued that it is possible to reduce the perceived complexity of specifications and to sustain the overall comprehension process. Specifications are complex buildings and the motivation for this work has been to diminish the problems that arise when trying to understand and maintain specifications.

The discussion of reasons for comprehension problems showed that specification comprehension is impeded by the overall complexity of the specification. Two aspects contributing to complexity have been discovered:

1. The size of the specification contributes a lot to the perceived complexity of the problem at hand.

2. There are too few clues for the reconstruction of the original structure and behavior that the specifier had in mind.

As complexity of size is the most critical type of complexity, it is promising to focus on size and abridge the specification for a problem at hand. For this reason the objective of this work has been to find answers to the following three questions:

Q1 What kind of abridgements are practicable? In other words, abstractions that sustain the comprehension process are to be identified.

Q2 How can those abridgements be achieved? The generation of abstractions has to be effective and efficient.

Q3 Can a reduction of complexity be expounded? A reduction, if any, has to be measured and it should be clear what kind of abstraction reduces complexity and to what extent.

In order to solve these problems, it is suggested to apply program comprehension approaches to specifications (Chap. 3). This includes the following two fields: The *generation of partial specifications* and the visualization of the *structure and hidden dependencies*.

For the identification of partial specifications a common understanding of terms and existing approaches is necessary.

First, a common terminology is presented (Chap. 4). It denotes those elements a specification is constructed of. There are several classes of such elements. On the syntactical level there are literals, primes and fragments. Literals are the primitives a specification is constructed of (e.g. identifiers or operators). Primes are basic semantic bearing entities (e.g. predicates), and fragments are simply compositions of primes. On the semantic level there are partial specifications which can be classified as specification chunks and slices. Slices and chunks are well-established concepts used in the field of program comprehension.

In this work it has been shown that specification slices and chunks are suitable concepts which support the comprehension process. As an answer to question Q1, it has been suggested to deal with slices and chunks as specification abridgements.

Dependency analysis turns out to be important when visualizing the structure and the behavior of the specification. It is also significant when calculating specification slices and chunks. For these two reasons this work introduces the notion of dependencies within specifications (also Chap. 4).

There is no execution order and almost no notion of control in declarative specification languages. Nevertheless, there are dependencies within specifications, dependencies which decide whether parts of a specification are to be evaluated or not. This work presents the definition of the notion of control dependencies in specifications. It is based on the simple idea that in specifications post-conditions are control dependent on pre-conditions. This reduces the problem of the identification of control dependencies to the problem of the identification of pre- and post-conditions.

Based on the notion of control this work provides definitions for control-, data- and syntactic dependencies in specifications. Upon this basis, specification slices and chunks are then defined.

However, not all specification languages make pre- and post-conditions as explicit as VDM. In Z, pre- and post-conditions have to be calculated by using pre-defined operations. Formally, a semantic analysis has to take place, but this is a time- and resource-consuming task which cannot be fully automated. Therefore it has been suggested to shorten the identification of pre- and post-conditions by looking at the use of identifiers in predicates. If there are identifiers that denote an after-state in a predicate, then the predicate is called a post-condition predicate. Otherwise, if there are no after-state identifiers, the predicate is called a pre-condition predicate.

For the visualization and calculation of dependencies one has to transform the specification to a network representation (Chap. 5). The net is called Augmented Specification Relationship Net (ASRN). It is designed to represent the loose structure of specification and thus exemplifies syntactic relationships and semantic dependencies. Vertices of the net contain value/definition/use information, and arcs represent the structural information and dependencies.

The transformation is bijective, thus, a backward transformation is possible, too. The net is also defined in a language-independent manner, however, the necessary transformation obviously depends on the specification language at hand. This work provides the transformation rules necessary to transform Z-specifications to the ASRN.

The transformation for Z is derived from the grammar of the specification language. The ASRN then provides the basis for an automated calculation of pre- and post conditions. With this, the net also enables the calculation of partial specifications, namely specification slices and chunks.

The ASRN (and the related transformation) can be used to generate effective abridgements of specifications. With this, question Q2 is answered.

Based on the ASRN, partial specifications are calculated efficiently. A small Java-based prototype has been implemented to transform Z specifications to an ASRN and to enable the generation of partial Z-specifications (Chap. 7). A reduction of size can be observed when applying the approach to existing specifications. The approach does not only reduce the size-complexity of the specification, but it has also effects on the logical complexity. Chap. 8 looks at these effects in more detail.

Partial specifications are generally smaller than the original specifications. There are several attributes which contribute to complexity. To measure the extent of the reduction of complexity, this work applies well-known complexity measures to specifications (Chap. 6).

Analogous to the measure of delivered lines of source instructions the number of primes are taken as a measure of size. In this thesis this measure is called Conceptual Complexity (CC for short).

The logical complexity is often measured by looking at dependencies. Here, the notions of the extended cyclomatic complexity and the definition/use count metric are assigned to specifications. The first looks at the number of control dependencies that are existent in the specification, the second looks at the number of data dependencies in the specification.

These measures are applied to four specifications of different (and increasing) size. For these specifications, several partial specifications are calculated, and case studies demonstrate the effects of partial specifications (Chap. 8). The following three hypotheses have been confirmed:

H1 Specification chunks reduce complexity to a greater extent than specification slices do.

H2 The effect obtainable by slicing and chunking rises with the size of the specification under consideration.

H3 The effect obtainable by slicing and chunking is significant in terms of reducing logical and conceptual complexity.

This analysis demonstrates that a reduction of complexity can be achieved, and that the approach is effective and efficient. This answers question Q3.

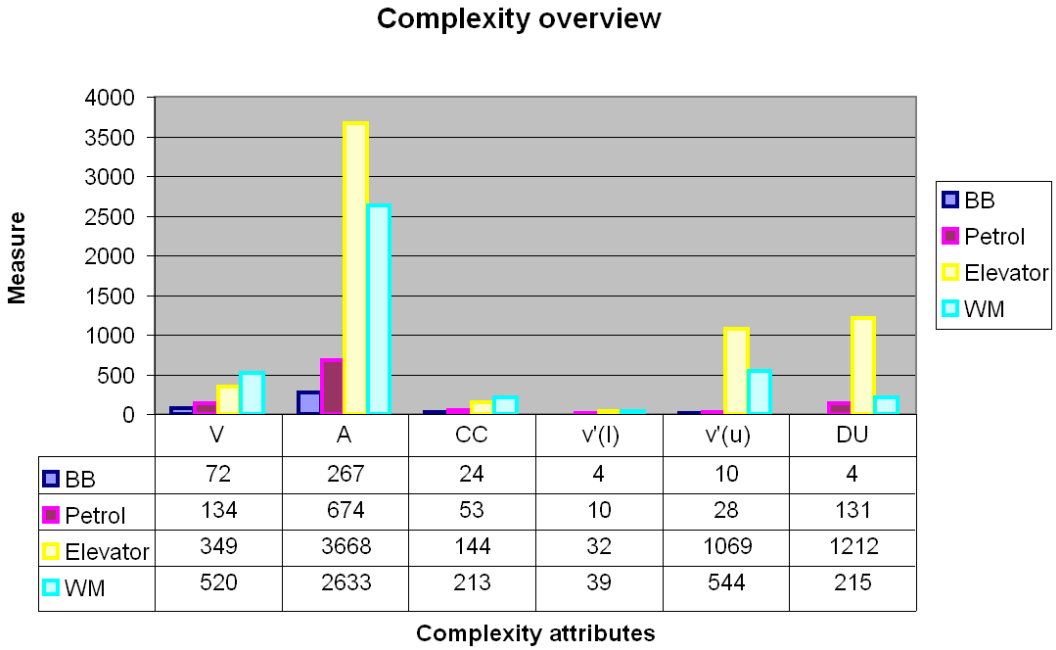
This thesis shows that partial specifications (specification slices and specification chunks) reduce the perceived (logical and cognitive) complexity of specifications. The effect of the approach is measurable, and it has been evaluated by making use of several specifications to be found in literature. The effect is amplified by visualizing hidden dependencies, dependencies which are used to calculate partial specifications anyway.

APPENDIX

A. EVALUATION MEASURES

This chapter summarizes the results of the studies describes in Chap. 8. For all primes in the axiomatic part of a Z paragraph, slices and two types of chunks (one containing control dependencies, one containing data-dependencies) were calculated. The set of identifiers was not restricted when calculating the chunk $SChunk(n, \Sigma_v, \{S, D\})$. Thus all data dependencies were taken into consideration. The measures are described in detail in Chaps. 8.1.2 and 6.3. The list of points of interest (POIs) can be found in App. A.5.

A.1 Comparison of Complexity and Influence of Size



	V	A	CC	v'(I)	v'(u)	DU
BB	72	267	24	4	10	4
Petrol	134	674	53	10	28	131
Elevator	349	3668	144	32	1069	1212
WM	520	2633	213	39	544	215

Fig. A.1: Comparison of four different specifications, the "Birthday-Book" specification (*BB*), the "Petrol-Station"-specification (*Petrol*), the "Elevator"-specification (*Elevator*), and the "ITC Window Manager"-specification (*WM*).

Spec.(Deps.)	V	A	k(V)	k(A)	k(CC)	k(v'(l))	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**		

Fig. A.2: Table summarizing the extent of reduction k in respect to four different specifications and three different types of abstraction.

Spec.(Deps.)	V	A	CC	V'	A'	CC'	M
BB Total	**72**	**267**	**24**				
BB (SCD)				47,143	148,143	13,00	
BB (SD)				40,714	114,571	9,29	
BB (SC)				37,857	117,000	10,43	
Mean				**41,905**	**126,571**	**10,905**	
Increase I	1,00	1,00	1,00				1,00
Approx.				41,905	126,571	10,905	
Delta				**1,000**	**1,000**	**1,000**	**1,000**
Petrol Total	**134**	**674**	**53**				
Petrol (SCD)				95,000	445,429	36,38	
Petrol (SD)				84,619	343,667	29,48	
Petrol (SC)				32,381	79,429	7,81	
Mean				**70,667**	**289,508**	**24,556**	
Increase I	1,86	2,52	2,21				2,20
Approx.				77,989	319,510	24,081	
Delta				**0,906**	**0,906**	**1,020**	**0,944**
Elevator Total	**349**	**3668**	**144**				
Elevator (SCD)				289,059	2759,765	108,36	
Elevator (SD)				191,167	710,216	42,16	
Elevator (SC)				110,000	261,284	23,57	
Mean				**196,742**	**1243,755**	**58,029**	
Increase I	4,85	13,74	6,00				8,20
Approx.				203,122	1738,816	65,429	
Delta				**0,969**	**0,715**	**0,887**	**0,857**
WM Total	**520**	**2633**	**213**				
WM (SCD)				236,087	958,087	76,41	
WM (SD)				91,739	159,283	13,08	
WM (SC)				96,217	183,696	18,71	
Mean				**141,348**	**433,688**	**36,065**	
Increase I	7,22	9,86	8,88				8,65
Approx.				302,646	1248,174	96,780	
Delta				**0,467**	**0,347**	**0,373**	**0,396**

Fig. A.3: Table summarizing the mean values of vertices and arcs in the partial specifications. The factor I expresses the increase in size, *Approx*. describes the assumed size of the partial specification, and *Delta* expresses the ratio between the real and supposed sizes.

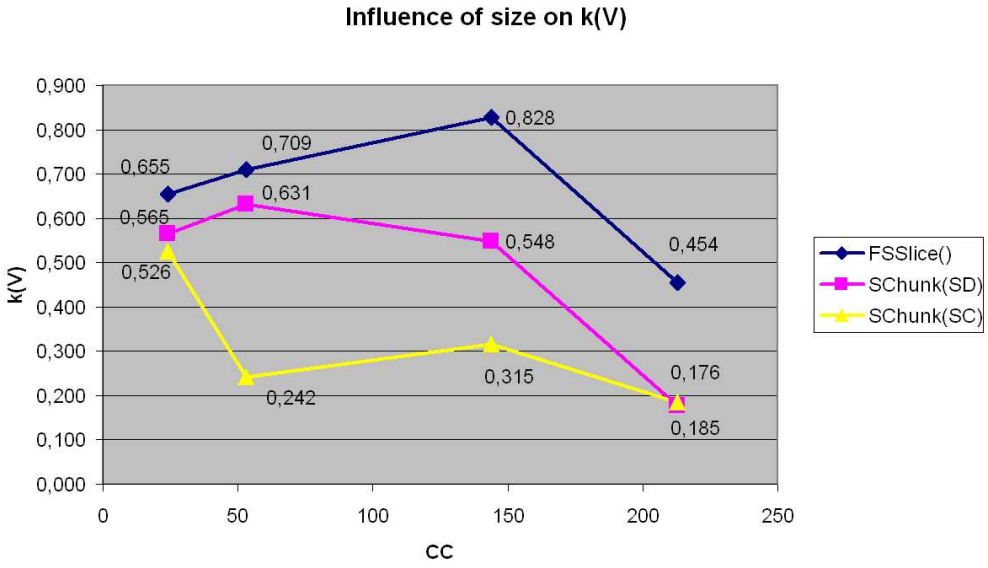


Fig. A.4: With increasing size of the specification, the number of vertices in the partial specification decreases.

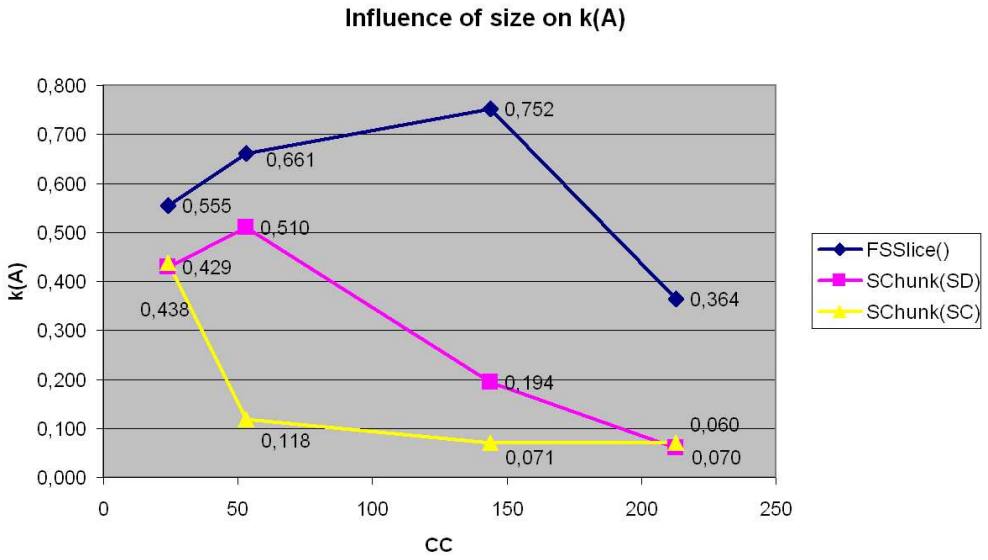


Fig. A.5: With increasing size of the specification, the number of arcs in the partial specification decreases.

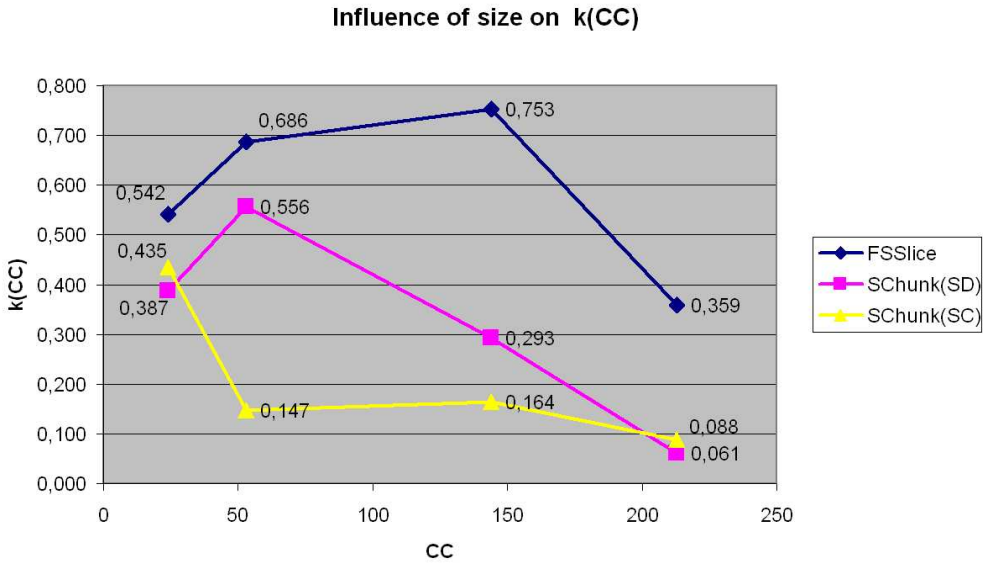


Fig. A.6: With increasing size of the specification, the conceptual complexity *CC* of the partial specification decreases.

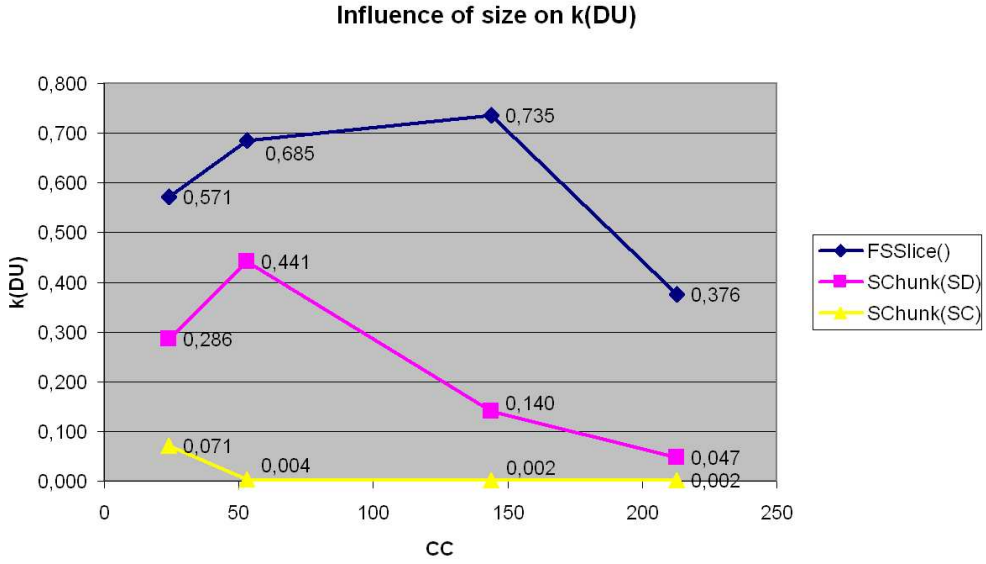


Fig. A.7: With increasing size of the specification, the definition/use count *DU* of the partial specification decreases.

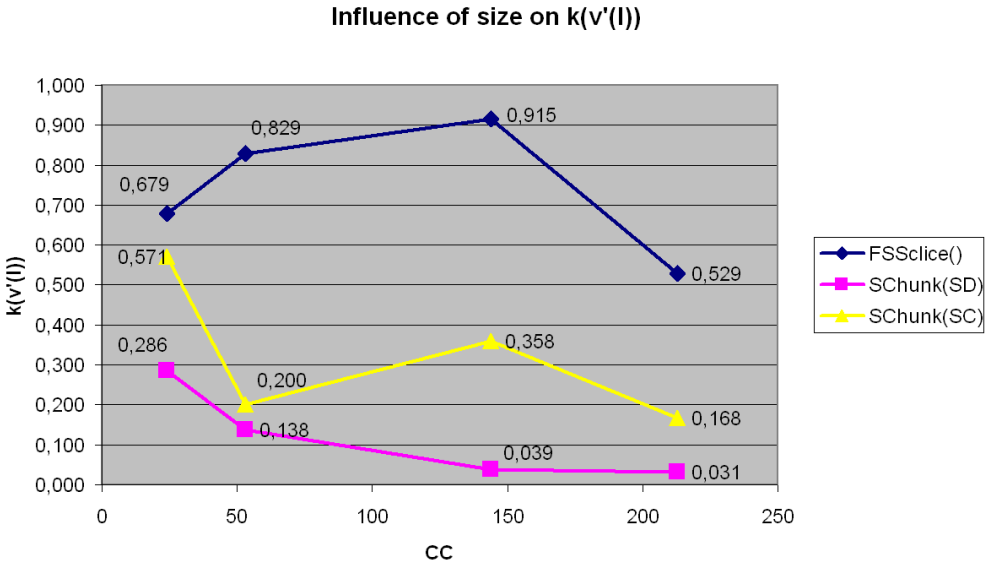


Fig. A.8: With increasing size of the specification, the lower bound of the extended cycl. complexity of the partial specification decreases.

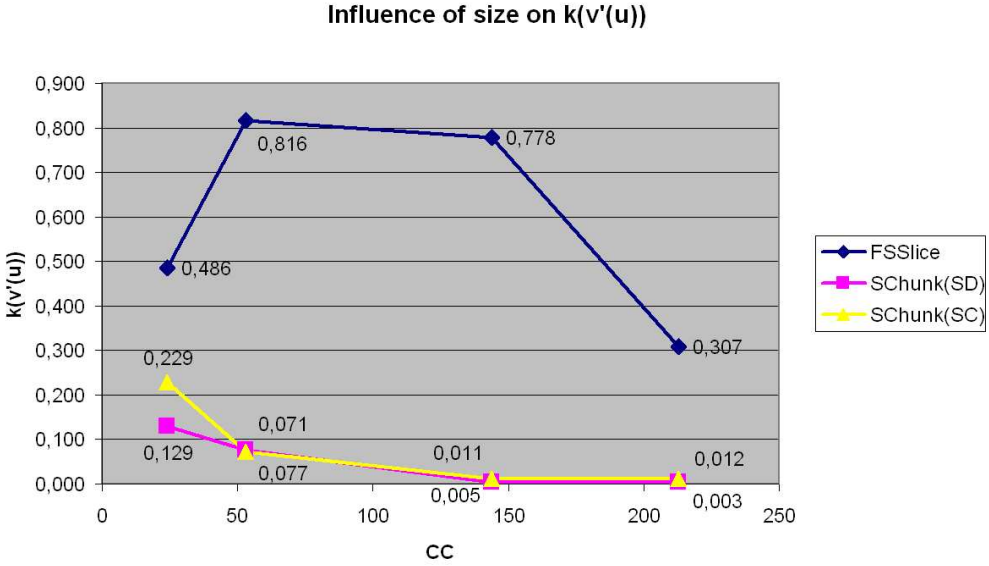


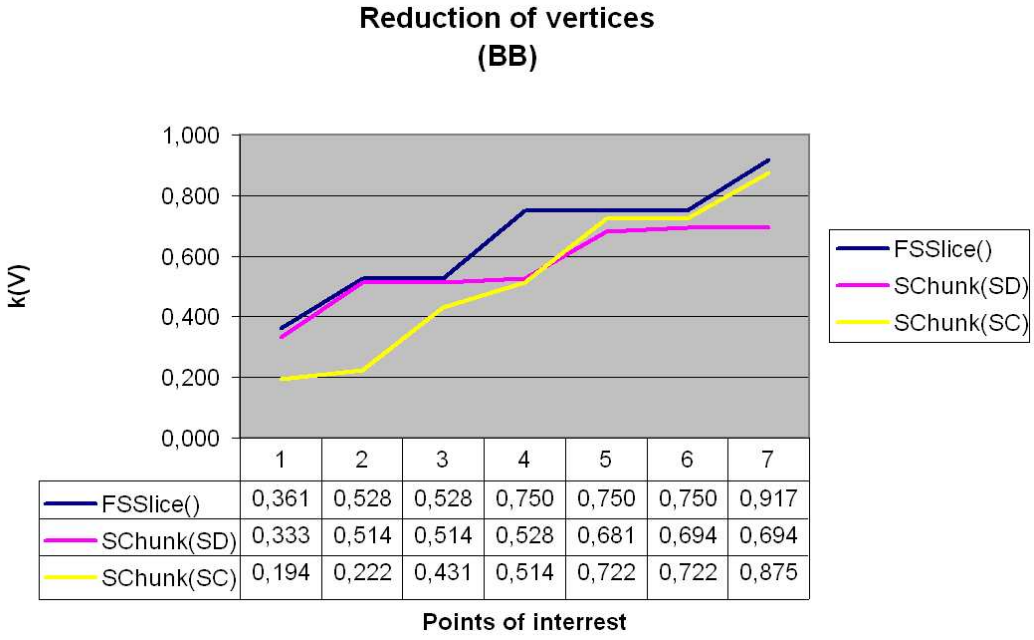
Fig. A.9: With increasing size of the specification, the upper bound of the extended cycl. complexity of the partial specification decreases.

A.2 Reduction of Complexity Attributes

This section visualizes the amount of reduction $r = 1/k$ of complexity for different specifications and their representation as an ASRN. Used complexity attributes are the number of vertices V and arcs A in the net, the conceptual complexity CC of the specification, the extended cyclomatic complexity $v' = (v(l), v(u))$, and the definition/use count metric (DU).

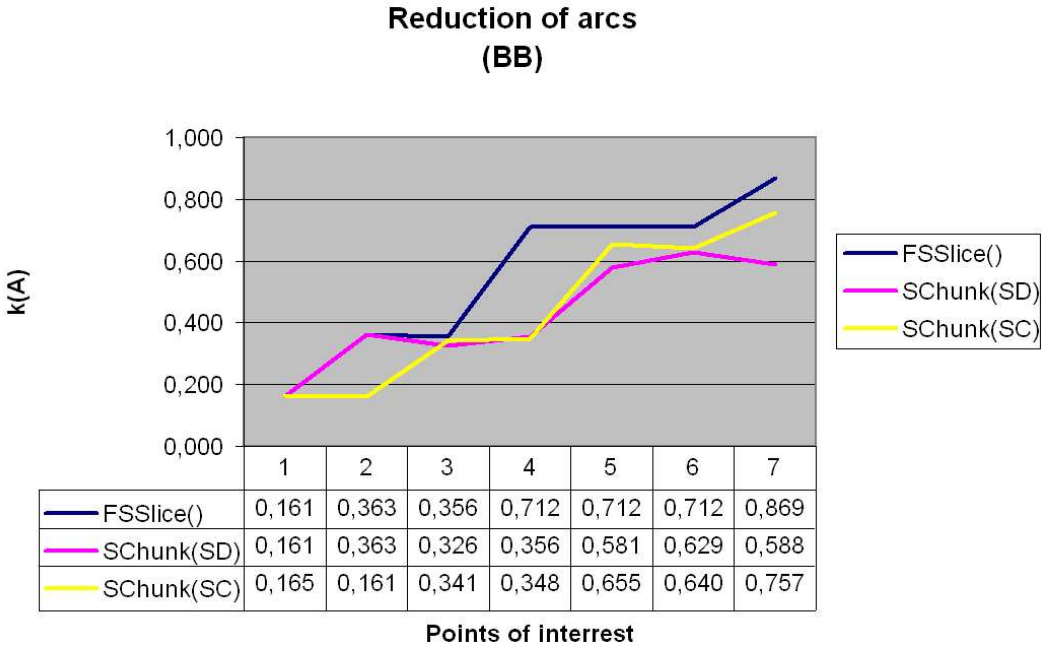
The following figures summarize the values for the reduction factor k, which is the result of calculating the ratio between the corresponding value of the partial specification and the value of the original specification. A lower value of k indicates higher reduction.

A.2.1 The Birthday Book Specification



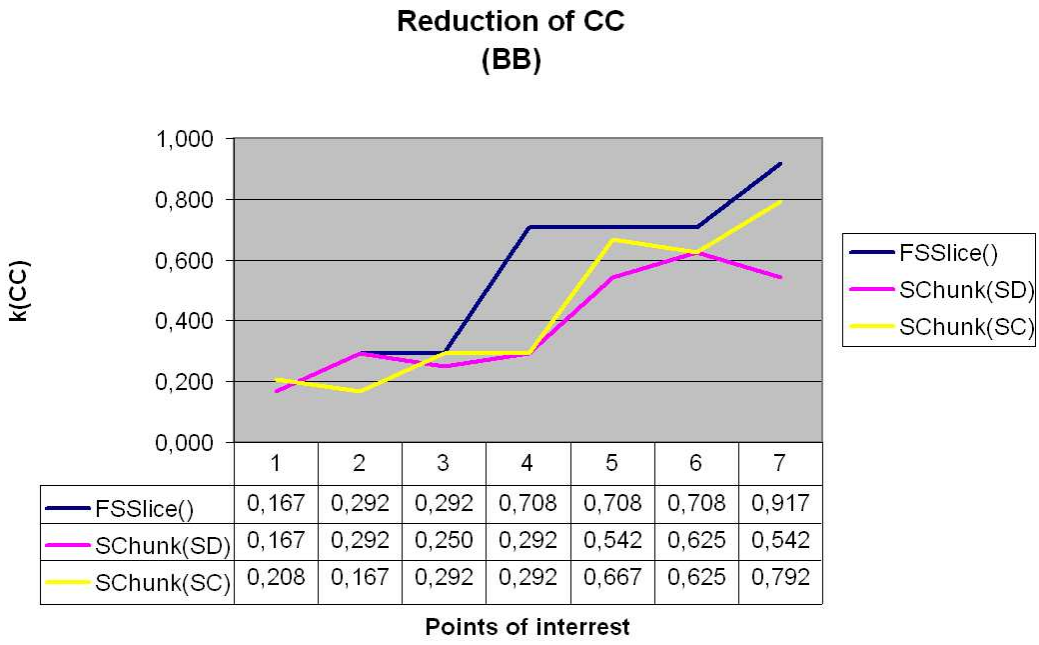
	1	2	3	4	5	6	7
FSSlice()	0,361	0,528	0,528	0,750	0,750	0,750	0,917
SChunk(SD)	0,333	0,514	0,514	0,528	0,681	0,694	0,694
SChunk(SC)	0,194	0,222	0,431	0,514	0,722	0,722	0,875

Fig. A.10: Extent of reduction of vertices for seven different points of interest and three different types of abstraction.



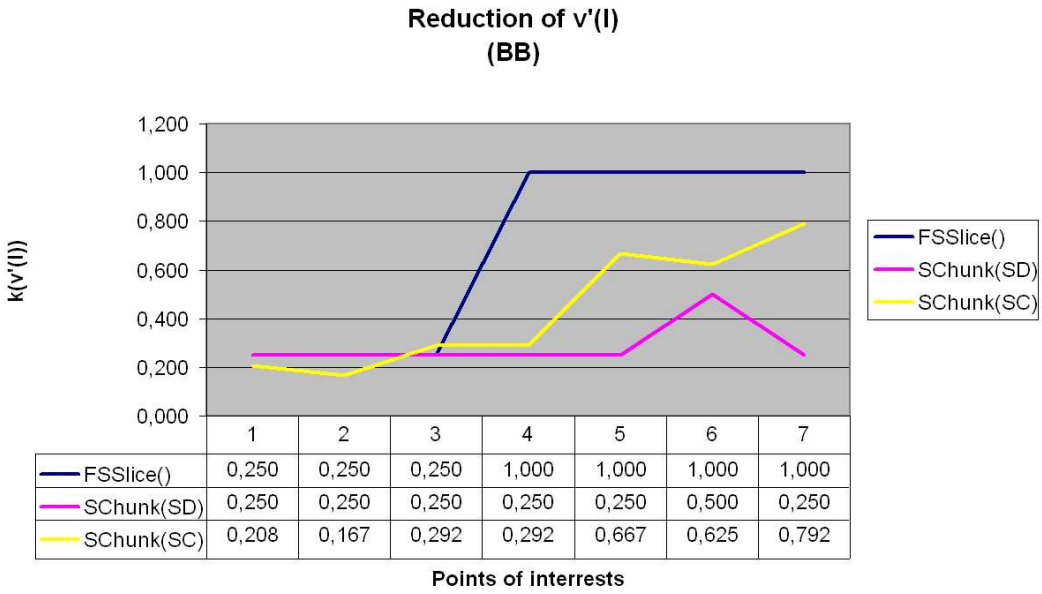
	1	2	3	4	5	6	7
FSSlice()	0,161	0,363	0,356	0,712	0,712	0,712	0,869
SChunk(SD)	0,161	0,363	0,326	0,356	0,581	0,629	0,588
SChunk(SC)	0,165	0,161	0,341	0,348	0,655	0,640	0,757

Fig. A.11: Extent of reduction of arcs for seven different points of interest and three different types of abstraction.



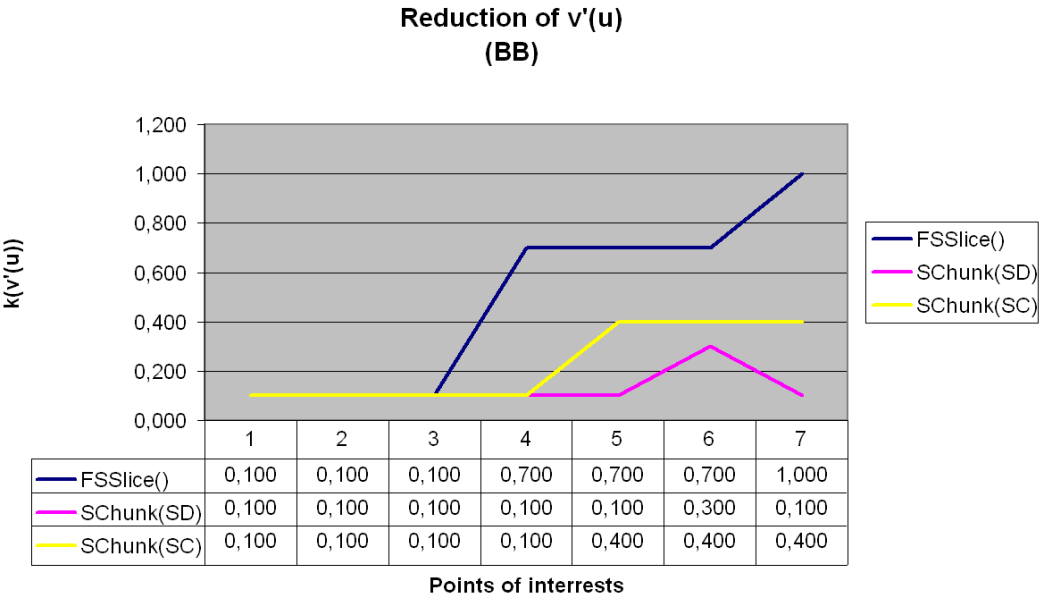
	1	2	3	4	5	6	7
FSSlice()	0,167	0,292	0,292	0,708	0,708	0,708	0,917
SChunk(SD)	0,167	0,292	0,250	0,292	0,542	0,625	0,542
SChunk(SC)	0,208	0,167	0,292	0,292	0,667	0,625	0,792

Fig. A.12: Extent of reduction of *CC* for seven different points of interest and three different types of abstraction.



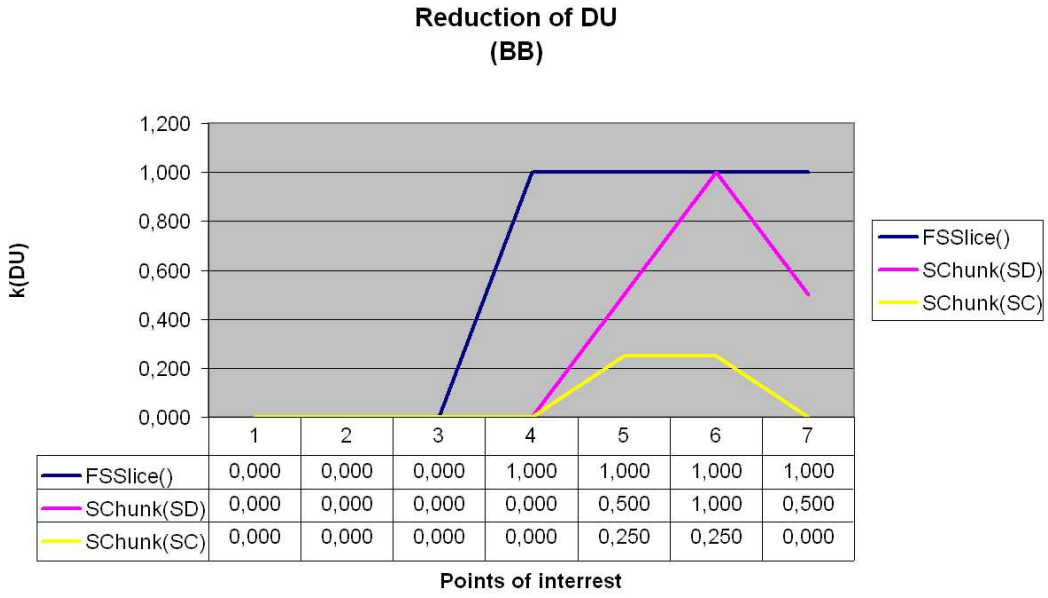
	1	2	3	4	5	6	7
FSSlice()	0,250	0,250	0,250	1,000	1,000	1,000	1,000
SChunk(SD)	0,250	0,250	0,250	0,250	0,250	0,500	0,250
SChunk(SC)	0,208	0,167	0,292	0,292	0,667	0,625	0,792

Fig. A.13: Extent of reduction of the lower bound of the extended cyclomatic complexity for seven different points of interest and three different types of abstraction.



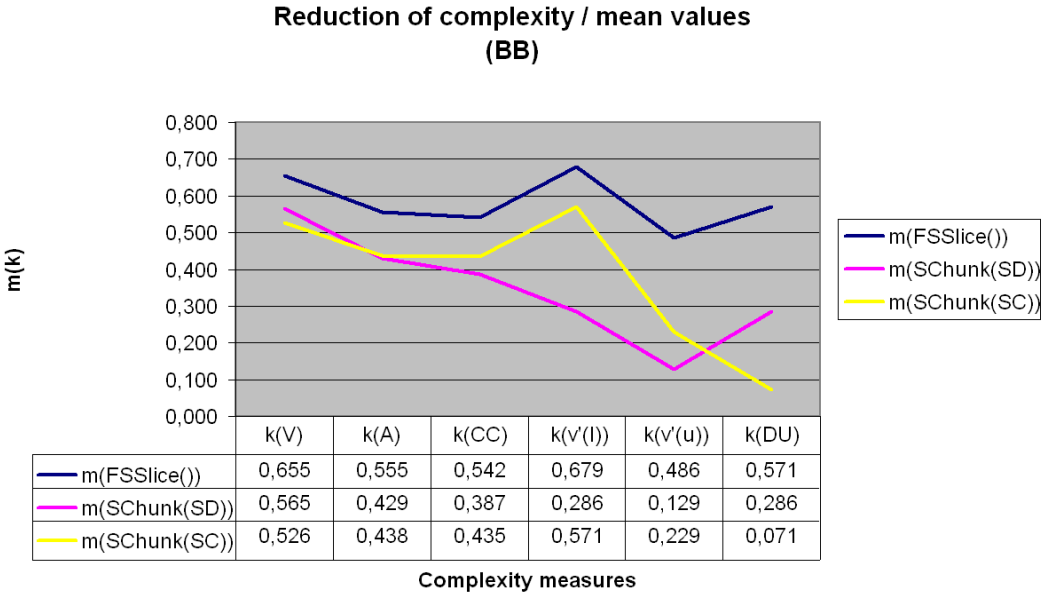
	1	2	3	4	5	6	7
FSSlice()	0,100	0,100	0,100	0,700	0,700	0,700	1,000
SChunk(SD)	0,100	0,100	0,100	0,100	0,100	0,300	0,100
SChunk(SC)	0,100	0,100	0,100	0,100	0,400	0,400	0,400

Fig. A.14: Extent of reduction of the upper bound of the extended cyclomatic complexity for seven different points of interest and three different types of abstraction.



	1	2	3	4	5	6	7
FSSlice()	0,000	0,000	0,000	1,000	1,000	1,000	1,000
SChunk(SD)	0,000	0,000	0,000	0,000	0,500	1,000	0,500
SChunk(SC)	0,000	0,000	0,000	0,000	0,250	0,250	0,000

Fig. A.15: Extent of reduction of *DU* for seven different points of interest and three different types of abstraction.



	k(V)	k(A)	k(CC)	k(v'(I))	k(v'(u))	k(DU)
m(FSSlice())	0,655	0,555	0,542	0,679	0,486	0,571
m(SChunk(SD))	0,565	0,429	0,387	0,286	0,129	0,286
m(SChunk(SC))	0,526	0,438	0,435	0,571	0,229	0,071

Fig. A.16: Summary of reduction of complexity attributes.

A.2.2 The Petrol Station Specification

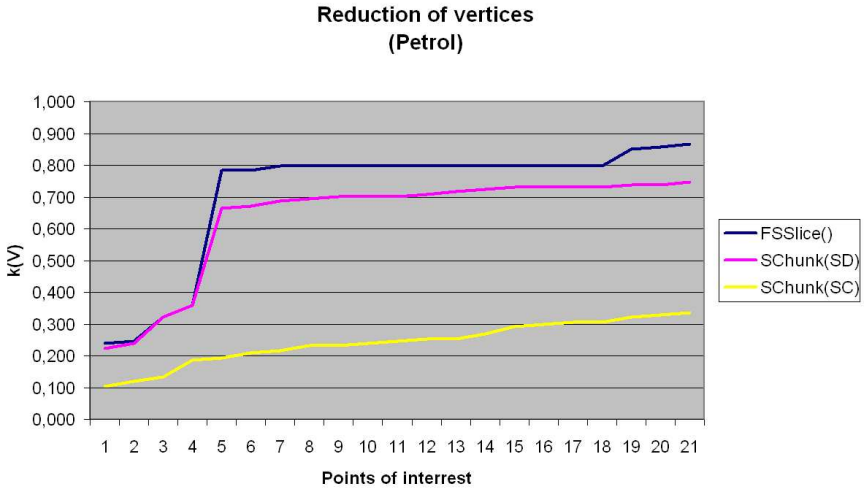


Fig. A.17: Extent of reduction of vertices for 21 different points of interest and three different types of abstraction.

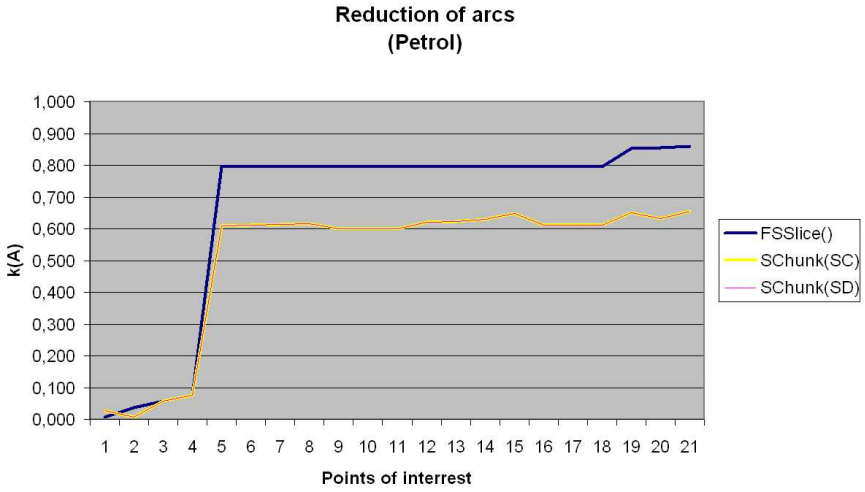


Fig. A.18: Extent of reduction of arcs for 21 different points of interest and three different types of abstraction.

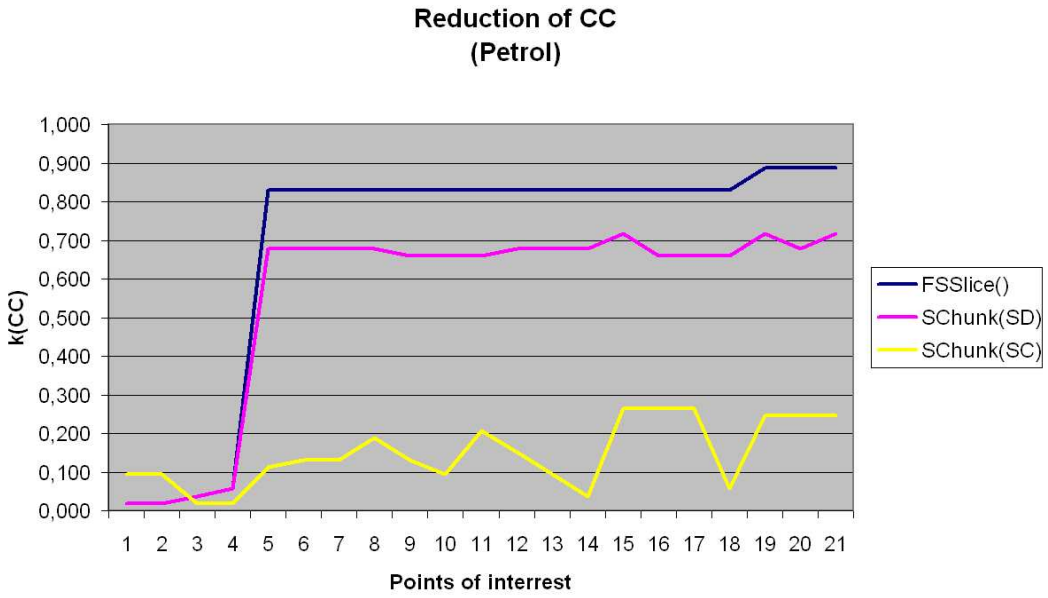


Fig. A.19: Extent of reduction of *CC* for 21 different points of interest and three different types of abstraction.

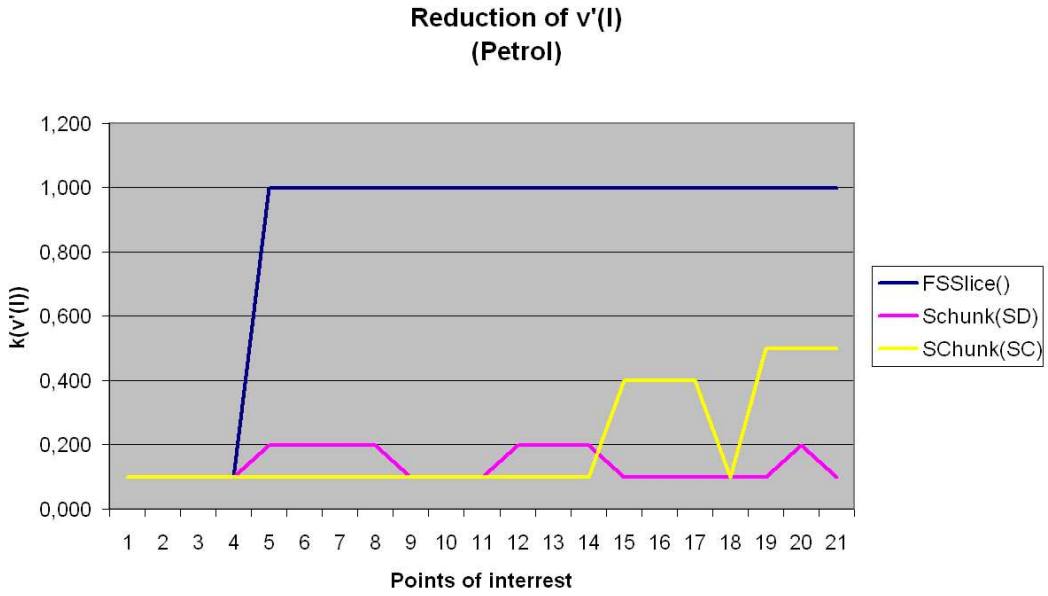


Fig. A.20: Extent of reduction of the lower bound of the extended cyclomatic complexity for 21 different points of interest and three different types of abstraction.

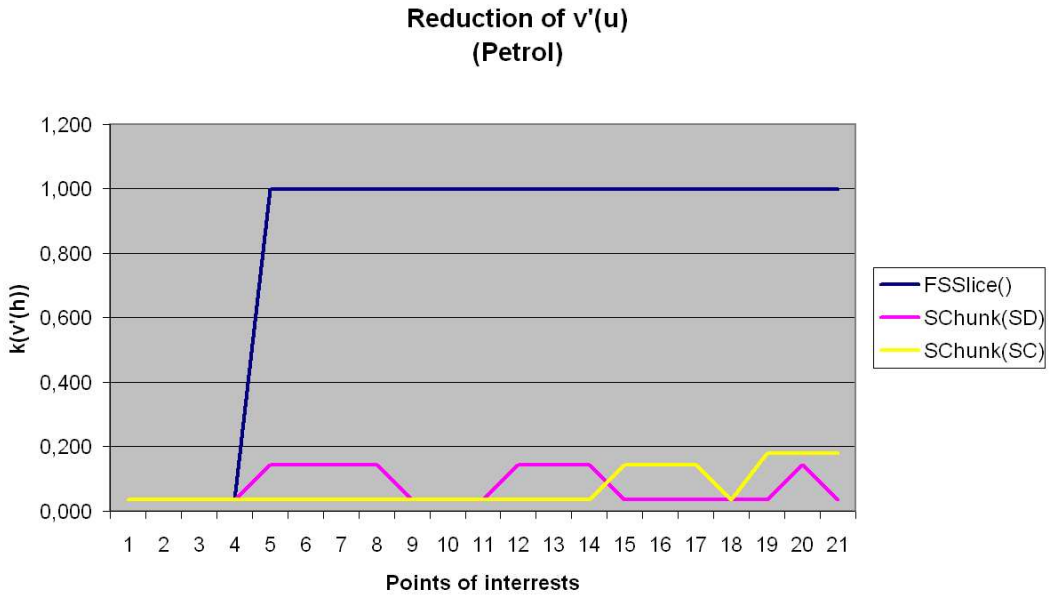


Fig. A.21: Extent of reduction of the upper bound of the extended cyclomatic complexity for 21 different points of interest and three different types of abstraction.

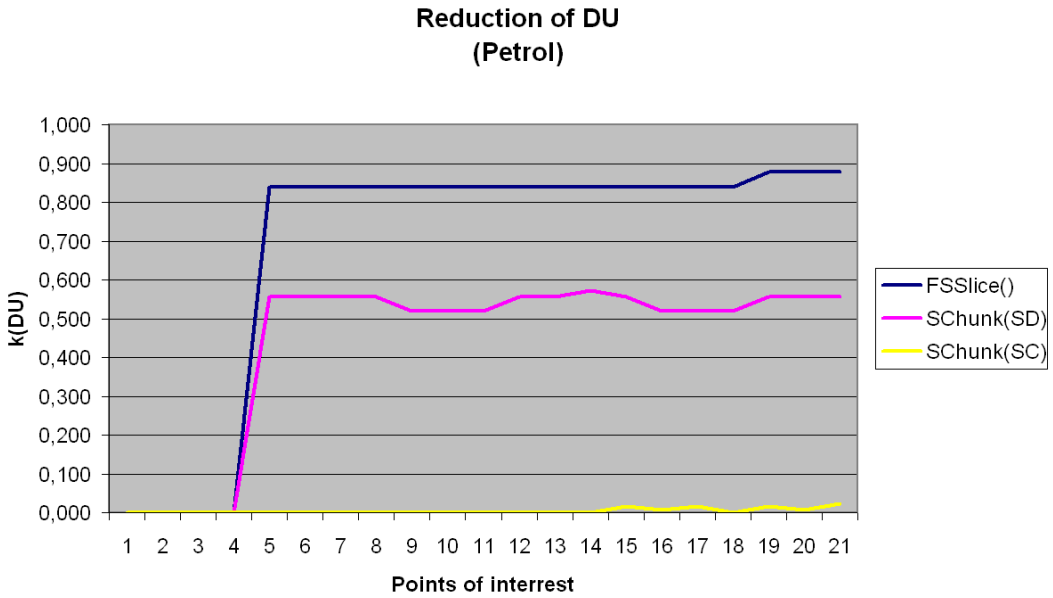
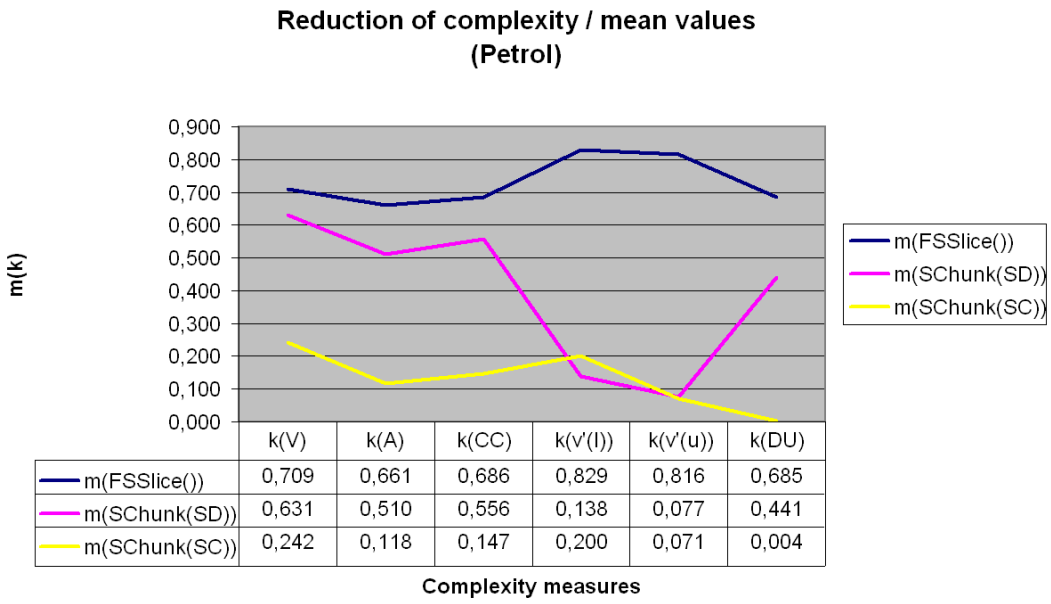


Fig. A.22: Extent of reduction of DU for 21 different points of interest and three different types of abstraction.



	k(V)	k(A)	k(CC)	k(v'(l))	k(v'(u))	k(DU)
m(FSSlice())	0,709	0,661	0,686	0,829	0,816	0,685
m(SChunk(SD))	0,631	0,510	0,556	0,138	0,077	0,441
m(SChunk(SC))	0,242	0,118	0,147	0,200	0,071	0,004

Fig. A.23: Summary of reduction of complexity attributes.

A.2.3 The ITC Window Manager Specification

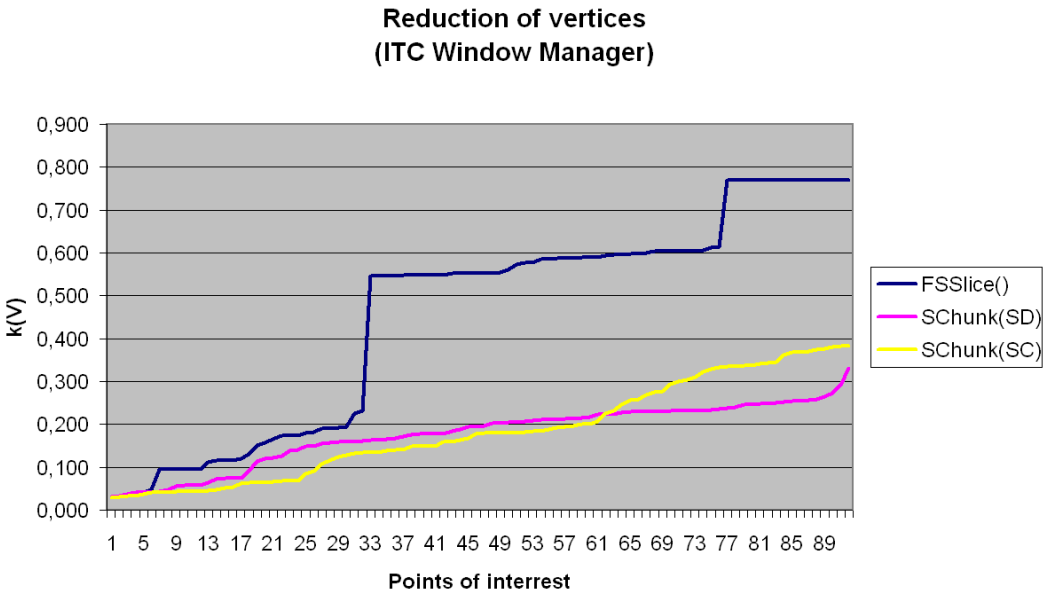


Fig. A.24: Extent of reduction of vertices for 92 different points of interest and three different types of abstraction.

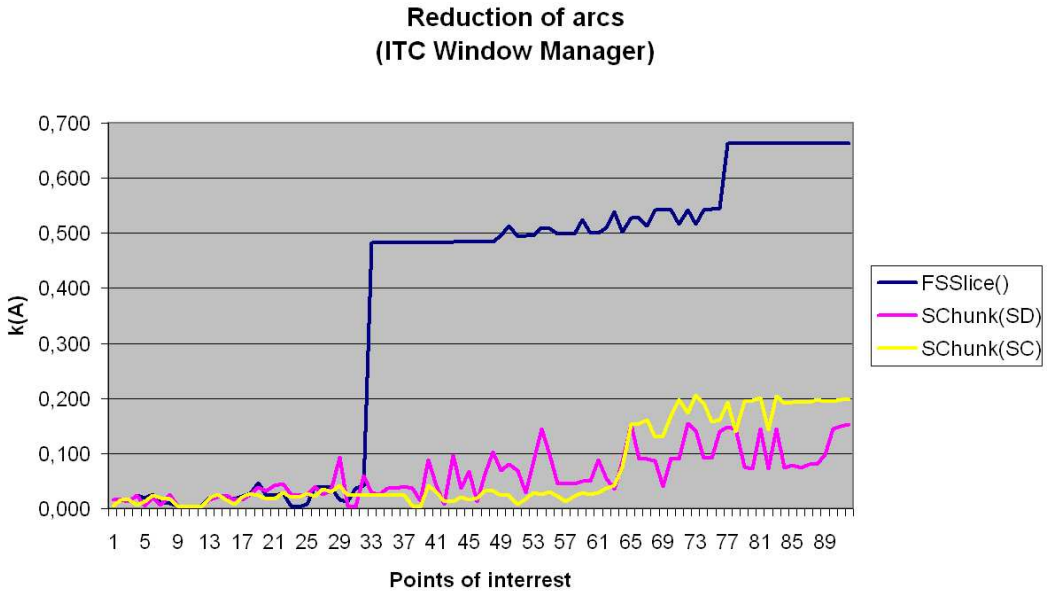


Fig. A.25: Extent of reduction of arcs for 92 different points of interest and three different types of abstraction.

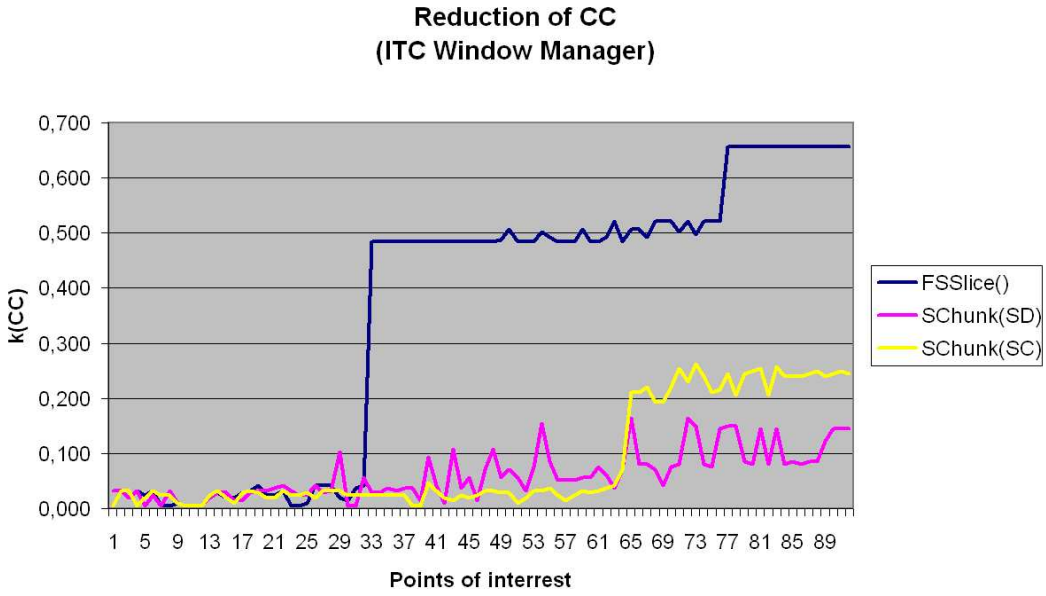


Fig. A.26: Extent of reduction of *CC* for 92 different points of interest and three different types of abstraction.

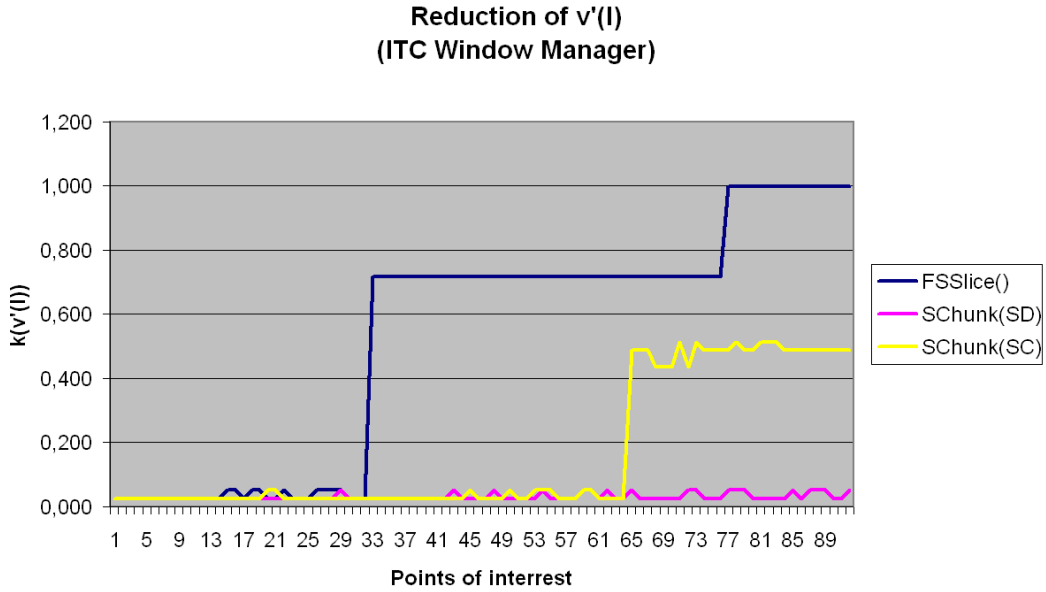


Fig. A.27: Extent of reduction of the lower bound of the extended cyclomatic complexity for 92 different points of interest and three different types of abstraction.

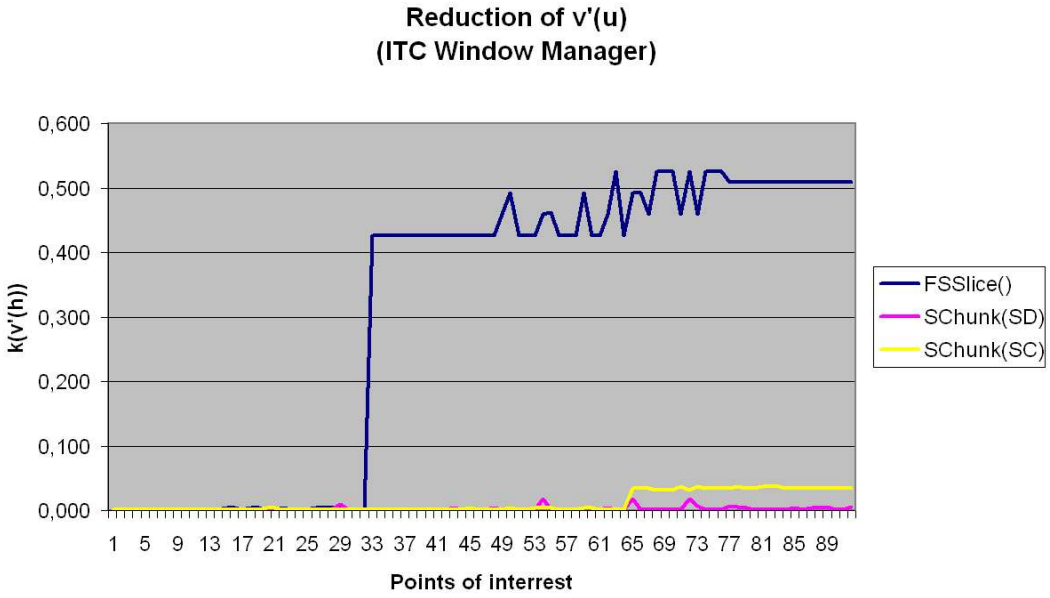


Fig. A.28: Extent of reduction of the upper bound of the extended cyclomatic complexity for 92 different points of interest and three different types of abstraction.

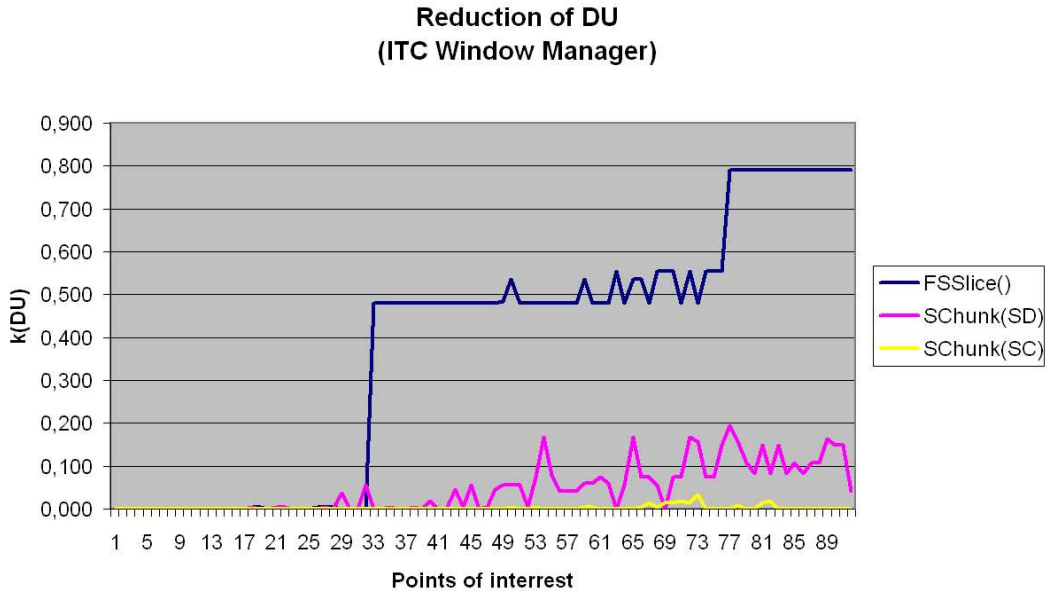
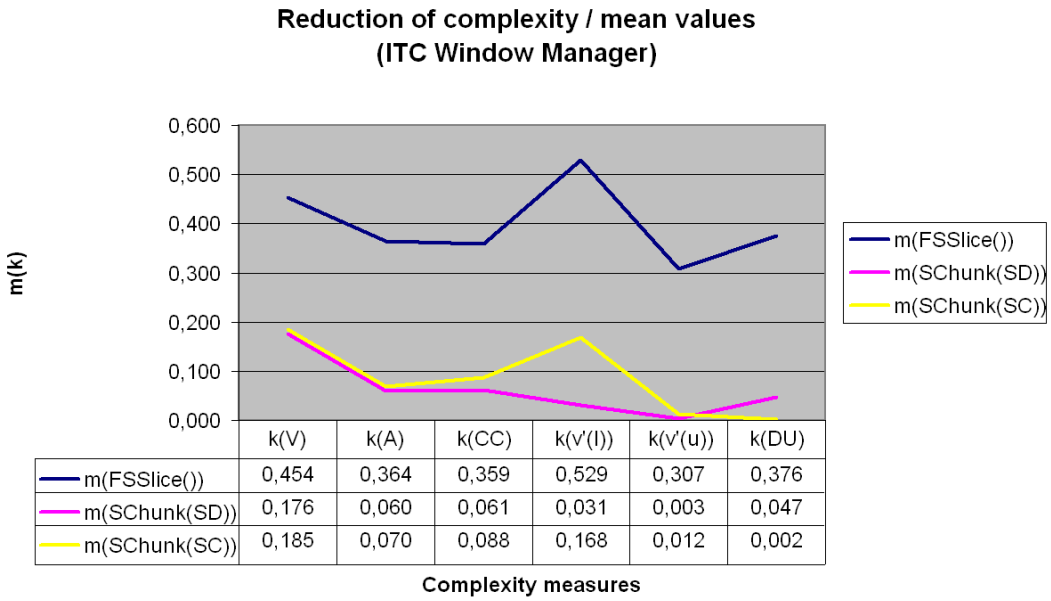


Fig. A.29: Extent of reduction of *DU* for 92 different points of interest and three different types of abstraction.



	k(V)	k(A)	k(CC)	k(v'(I))	k(v'(u))	k(DU)
m(FSSlice())	0,454	0,364	0,359	0,529	0,307	0,376
m(SChunk(SD))	0,176	0,060	0,061	0,031	0,003	0,047
m(SChunk(SC))	0,185	0,070	0,088	0,168	0,012	0,002

Fig. A.30: Summary of reduction of complexity attributes.

A.2.4 The Elevator Specification

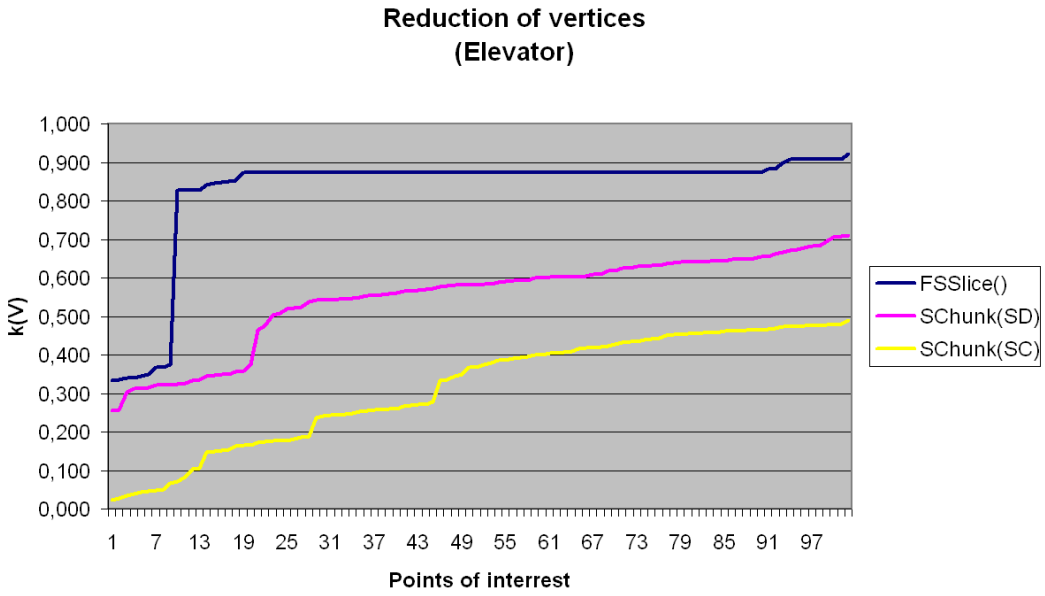


Fig. A.31: Extent of reduction of vertices for 102 different points of interest and three different types of abstraction.

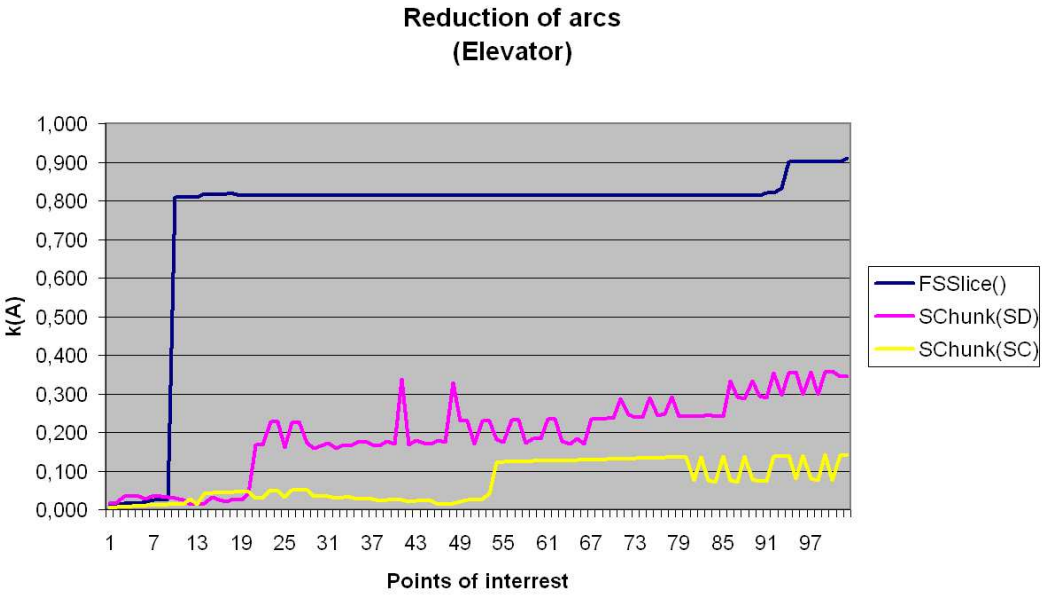


Fig. A.32: Extent of reduction of arcs for 102 different points of interest and three different types of abstraction.

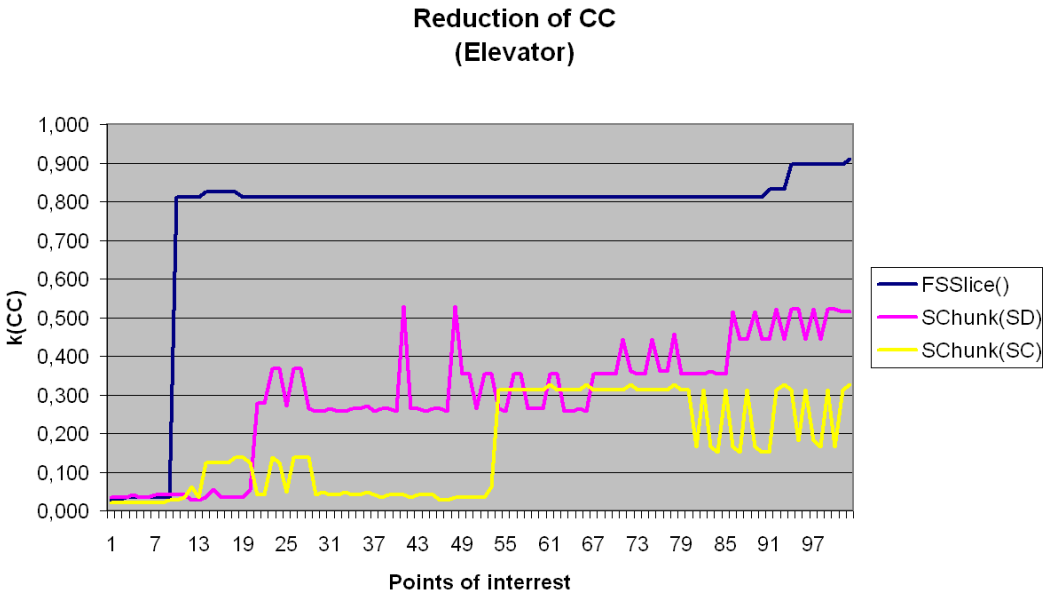


Fig. A.33: Extent of reduction of *CC* for 102 different points of interest and three different types of abstraction.

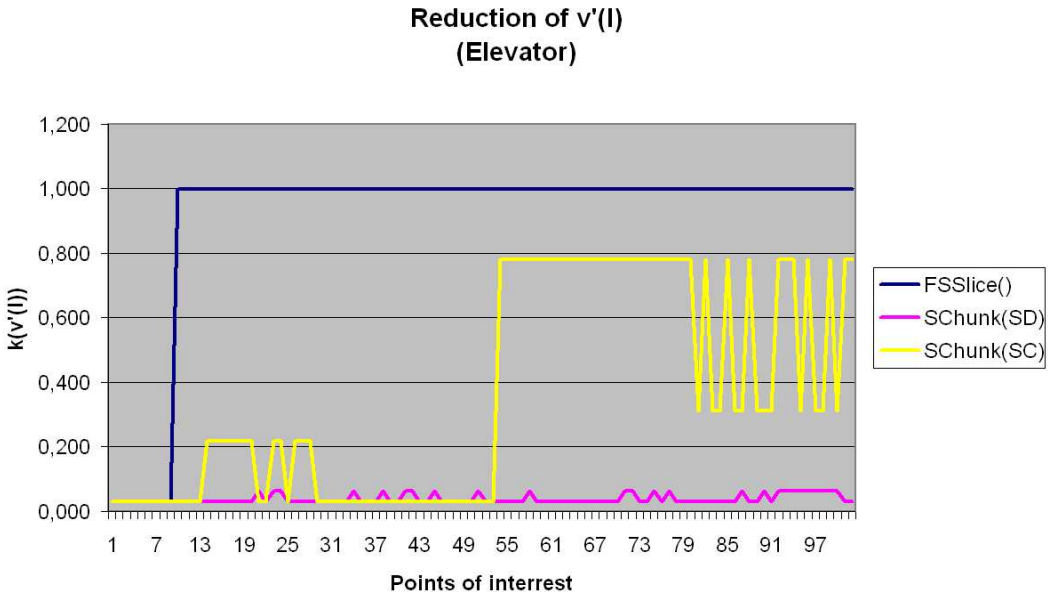


Fig. A.34: Extent of reduction of the lower bound of the extended cyclomatic complexity for 102 different points of interest and three different types of abstraction.

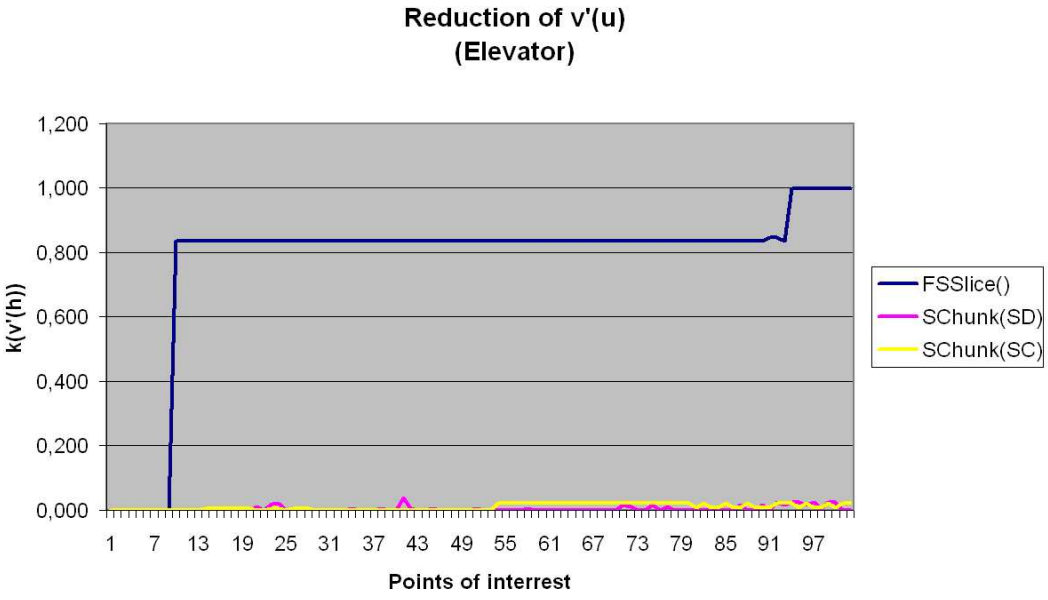


Fig. A.35: Extent of reduction of the upper bound of the extended cyclomatic complexity for 102 different points of interest and three different types of abstraction.

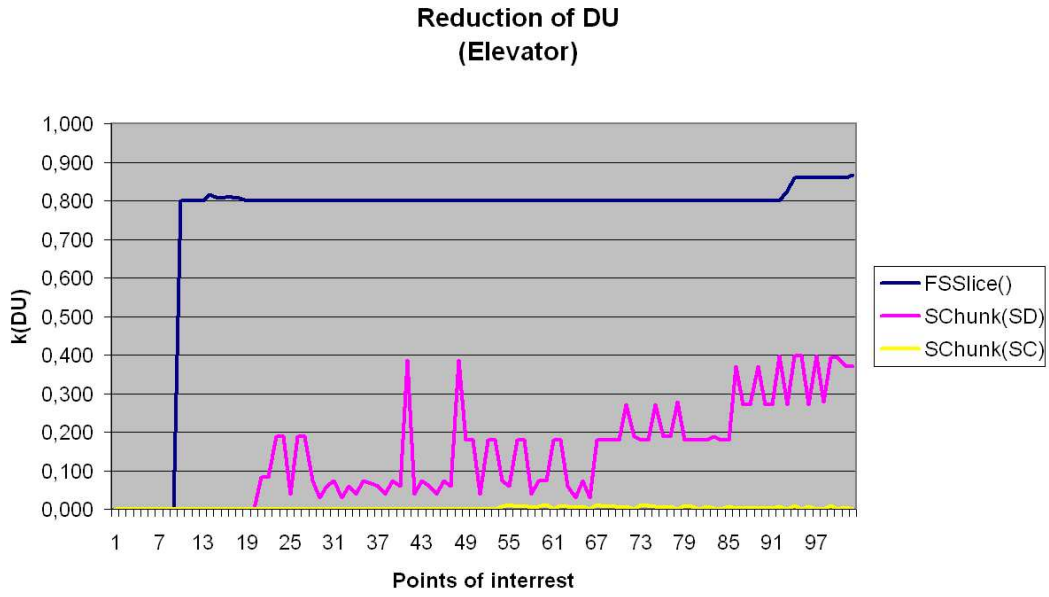
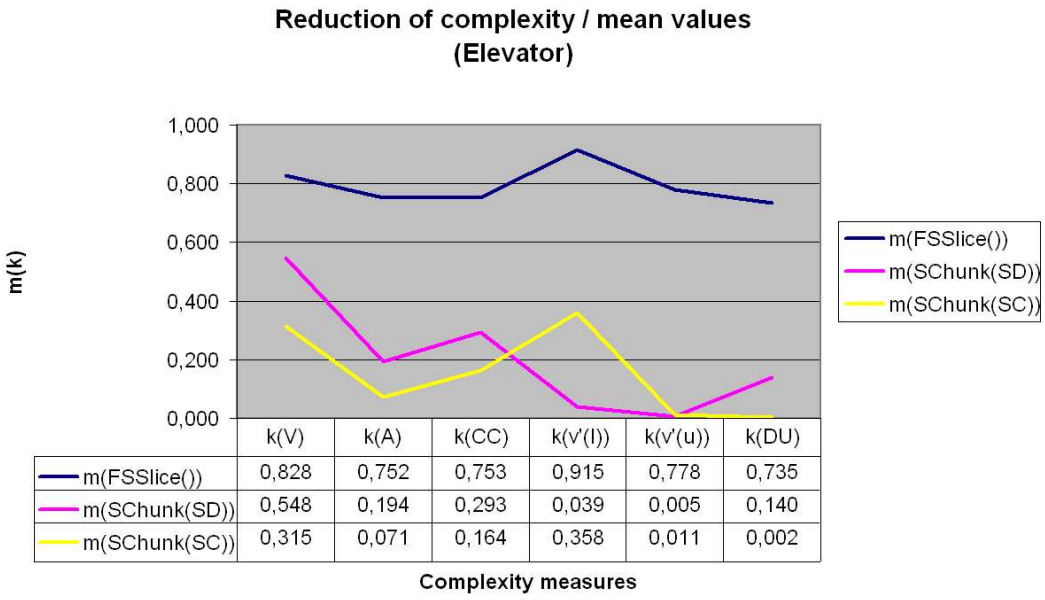


Fig. A.36: Extent of reduction of DU for 102 different points of interest and three different types of abstraction.

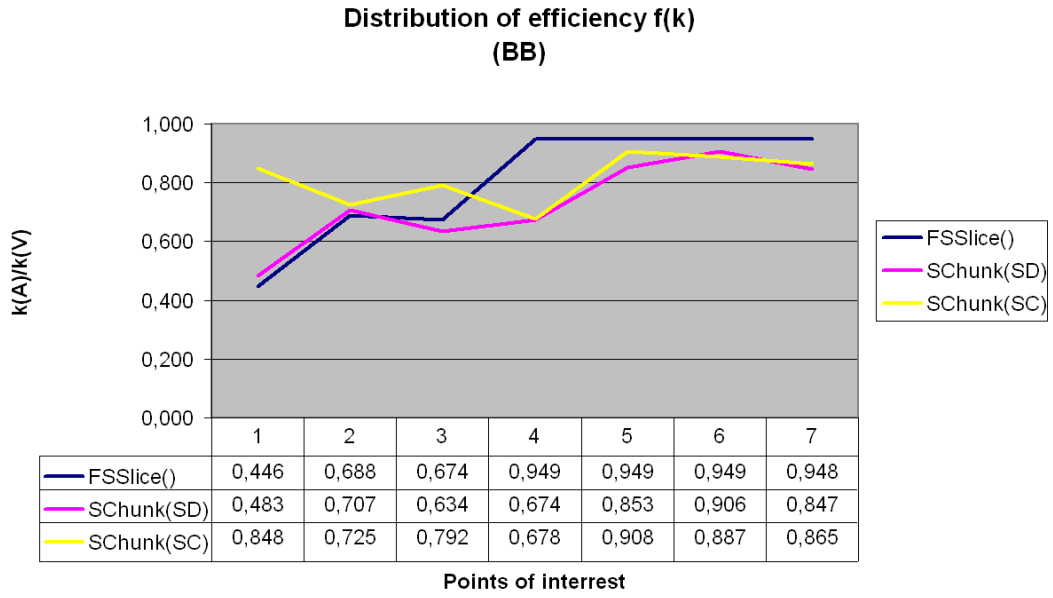


	k(V)	k(A)	k(CC)	k(v'(l))	k(v'(u))	k(DU)
m(FSSlice())	0,828	0,752	0,753	0,915	0,778	0,735
m(SChunk(SD))	0,548	0,194	0,293	0,039	0,005	0,140
m(SChunk(SC))	0,315	0,071	0,164	0,358	0,011	0,002

Fig. A.37: Summary of reduction of complexity attributes.

A.3 Efficiency of Partial Specifications

According to Chap 8.4, the efficiency of the approach can be described by the efficiency factor $f(k)$. It is calculated by dividing the reduction factor of arcs with the reduction factor of vertices ($f(k) = k(A)/k(V)$). If the value is lower than 1 then the number of arcs (and with them dependencies) decreases to a higher extent than the number of vertices in the net. The approach is said to be efficient in respect to logical complexity.



	1	2	3	4	5	6	7
FSSlice()	0,446	0,688	0,674	0,949	0,949	0,949	0,948
SChunk(SD)	0,483	0,707	0,634	0,674	0,853	0,906	0,847
SChunk(SC)	0,848	0,725	0,792	0,678	0,908	0,887	0,865

Fig. A.38: Variation of the efficiency factor $f(k)$ in respect to seven possible points of interest and three types of abstraction.

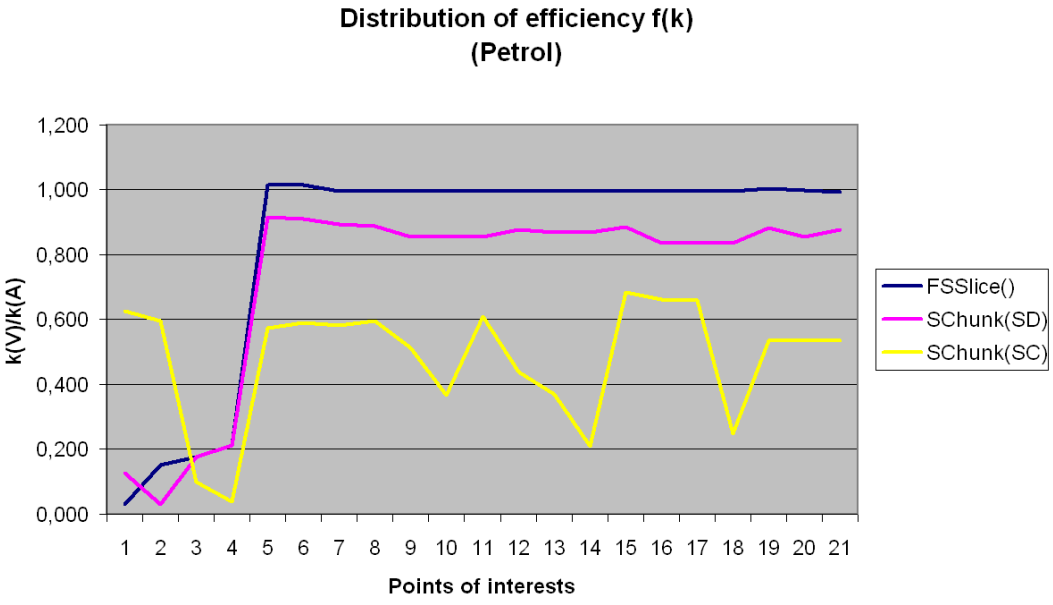


Fig. A.39: Variation of the efficiency factor $f(k)$ in respect to 21 possible points of interest and three types of abstraction.

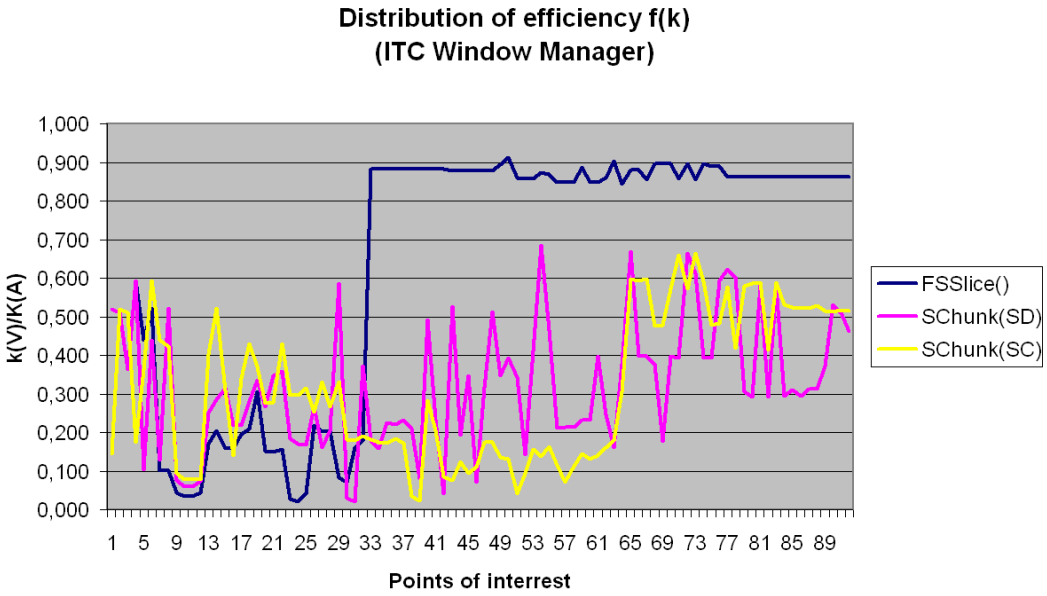


Fig. A.40: Variation of the efficiency factor $f(k)$ in respect to the 92 possible points of interest and three types of abstraction.

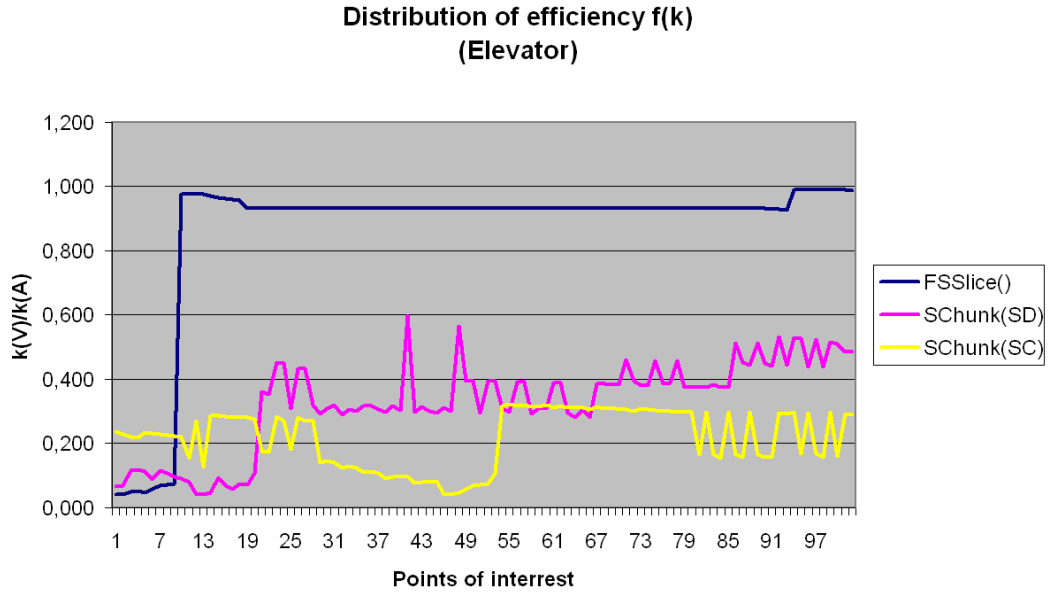


Fig. A.41: Variation of the efficiency factor $f(k)$ in respect to the 102 possible points of interest and three types of abstraction.

A.4 Influence of Size

Size strongly influences the efficiency of the approach. The larger the specifications are, the relatively smaller are the generated partial specifications. There is much more reduction when generating specification chunks. The following figures demonstrate that the reduction increases with increasing size of the specification.

For vertices V, arcs A, and the conceptual complexity CC it presents the increase of the values for the full specification, the values for different partial specifications, and the mean value summarizing the effect of the application of all types of abstraction.

Size does not only influence the number of vertices. It also influences the amount of reduction. Therefore this section summarizes the influence of the size of the specification on the dependent variables $k(V)$, $k(A)$, $k(CC)$, $k(v'(l))$, $k(v'(u))$, and $k(CC)$.

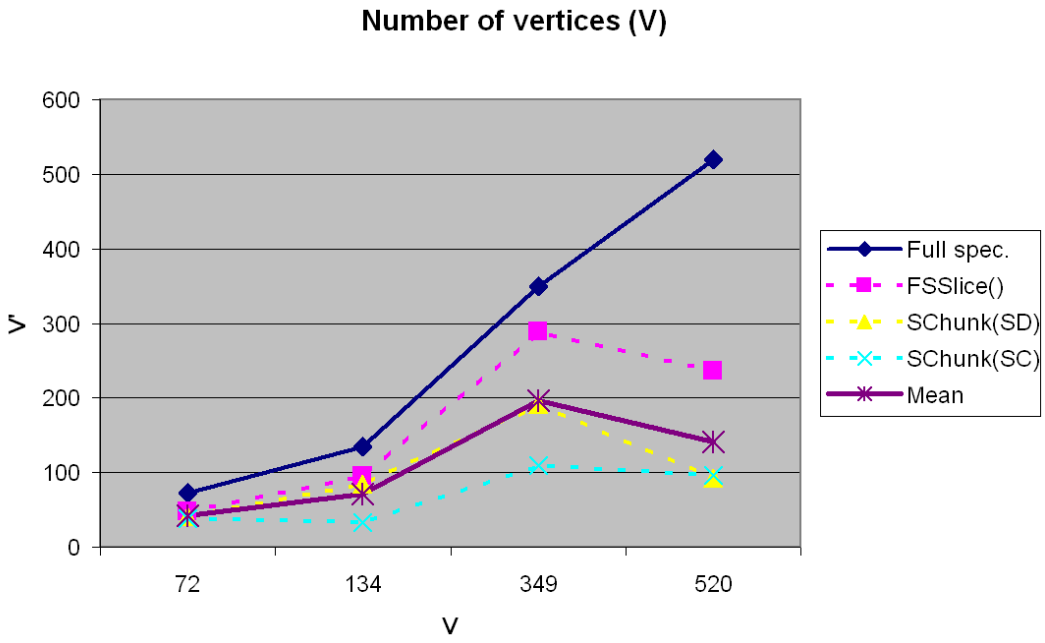


Fig. A.42: With the increasing size of specifications (described by V), the approach reduces the number of vertices to a much higher extent.

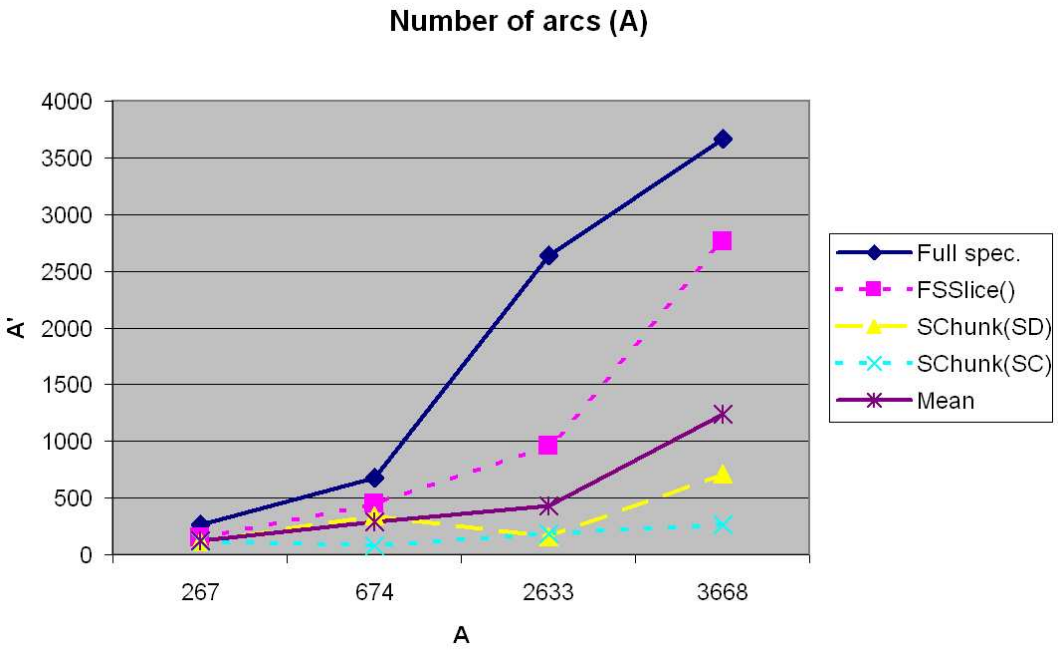


Fig. A.43: With the increasing size of specifications (described by A), the approach reduces the number of arcs to a much higher extent.

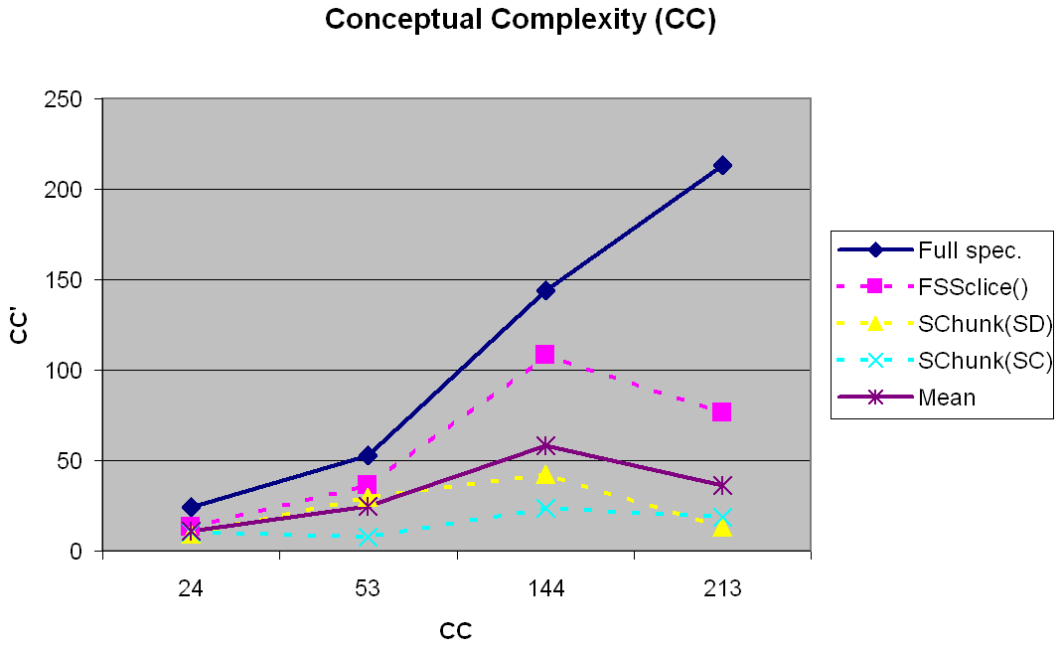


Fig. A.44: With the increasing size of specifications (described by CC), the approach reduces the number of CC to a much higher extent.

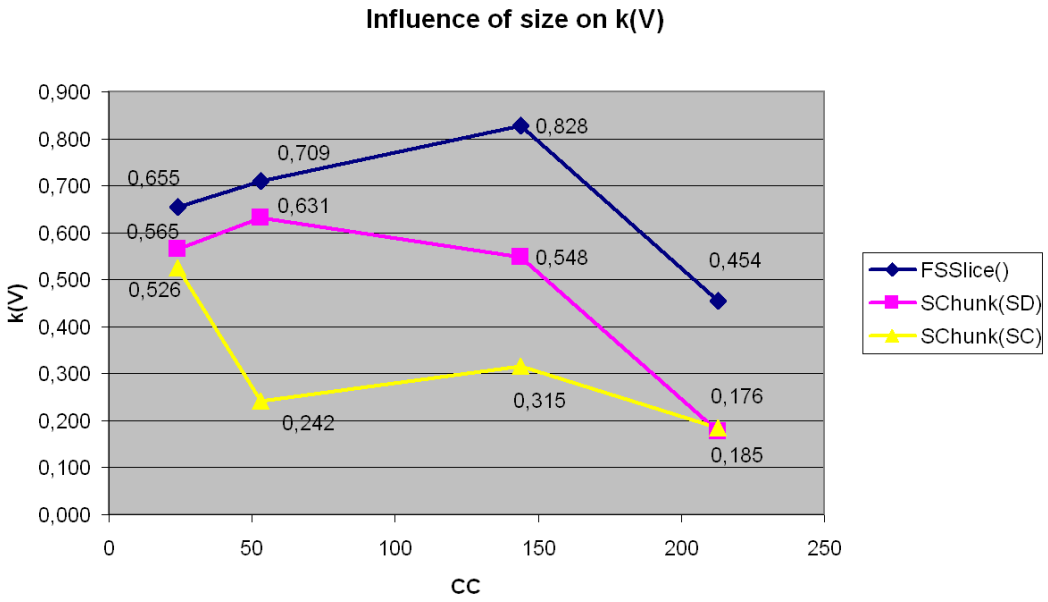


Fig. A.45: With the increasing size of specifications (described by CC), the approach reduces the number of V to a much higher extent.

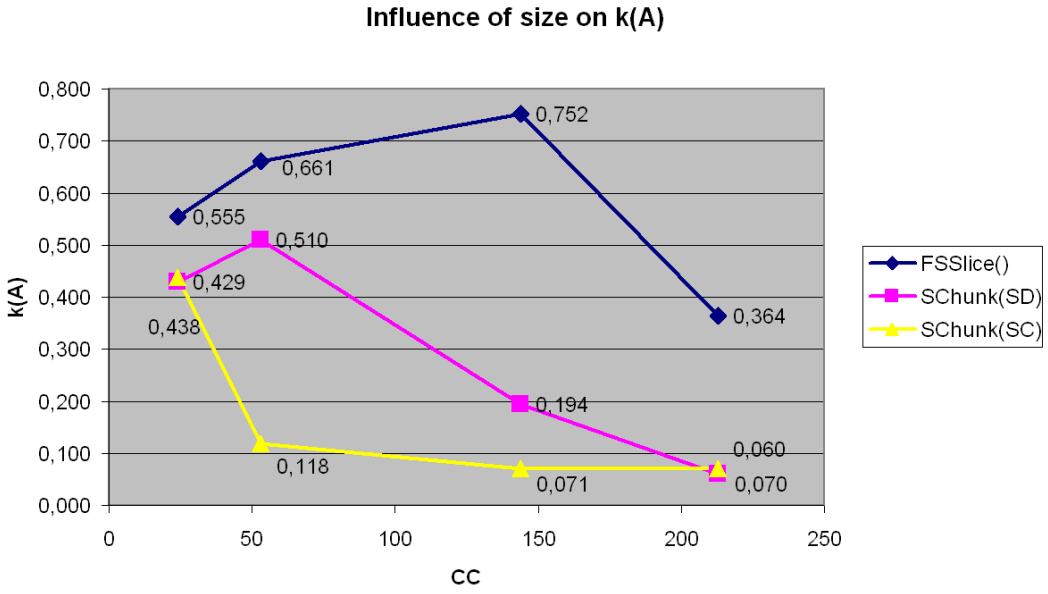


Fig. A.46: With the increasing size of specifications (described by CC), the approach reduces the number of A to a much higher extent.

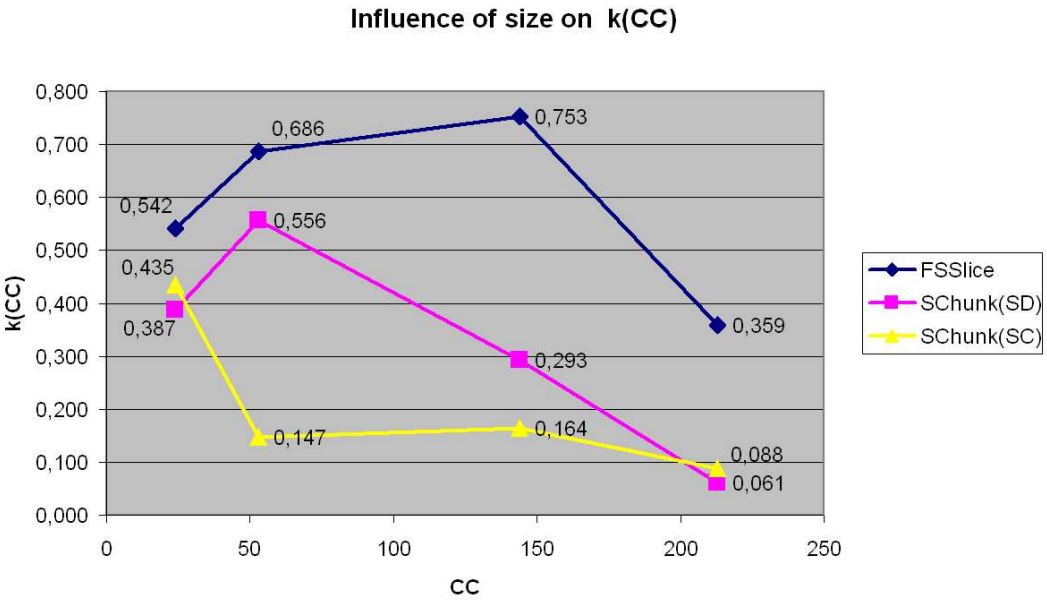


Fig. A.47: With the increasing size of specifications (described by CC), the approach reduces the number of CC to a much higher extent.

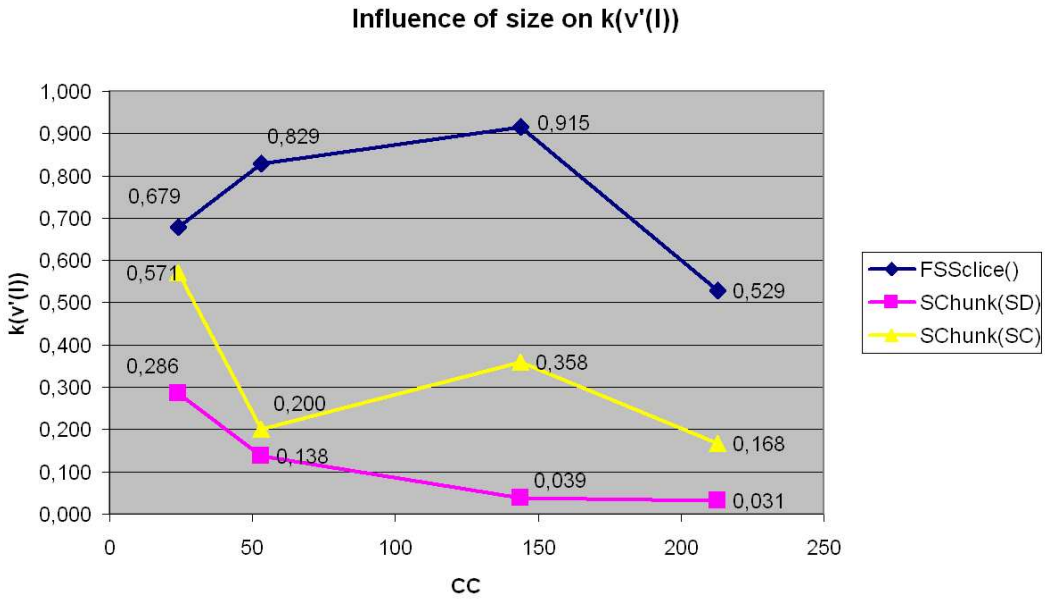


Fig. A.48: With the increasing size of specifications (described by CC), the approach reduces the number of $v'(l)$ to a much higher extent.

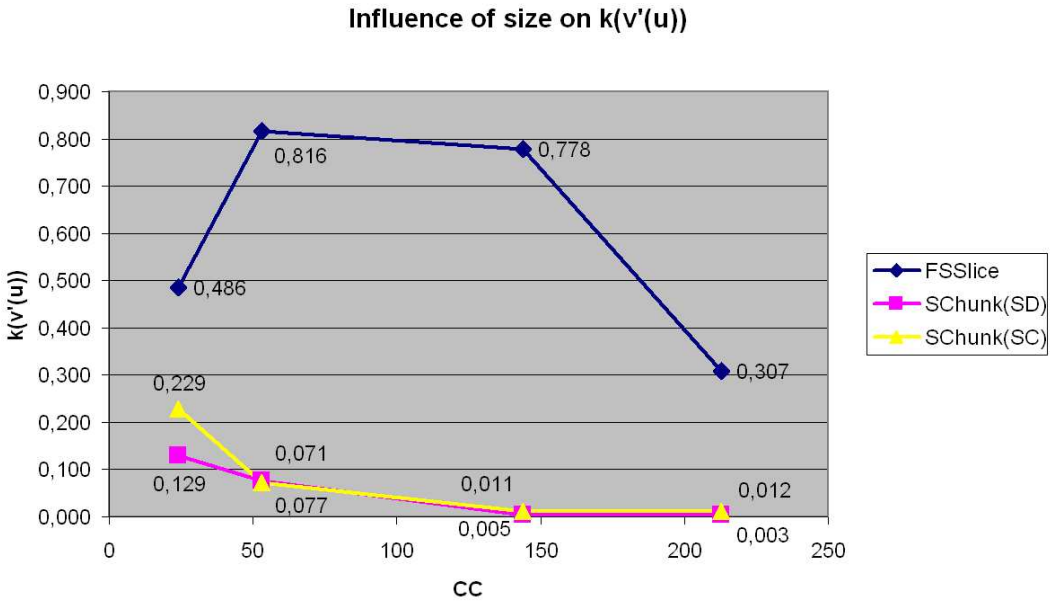


Fig. A.49: With the increasing size of specifications (described by CC), the approach reduces the number of $v'(u)$ to a much higher extent.

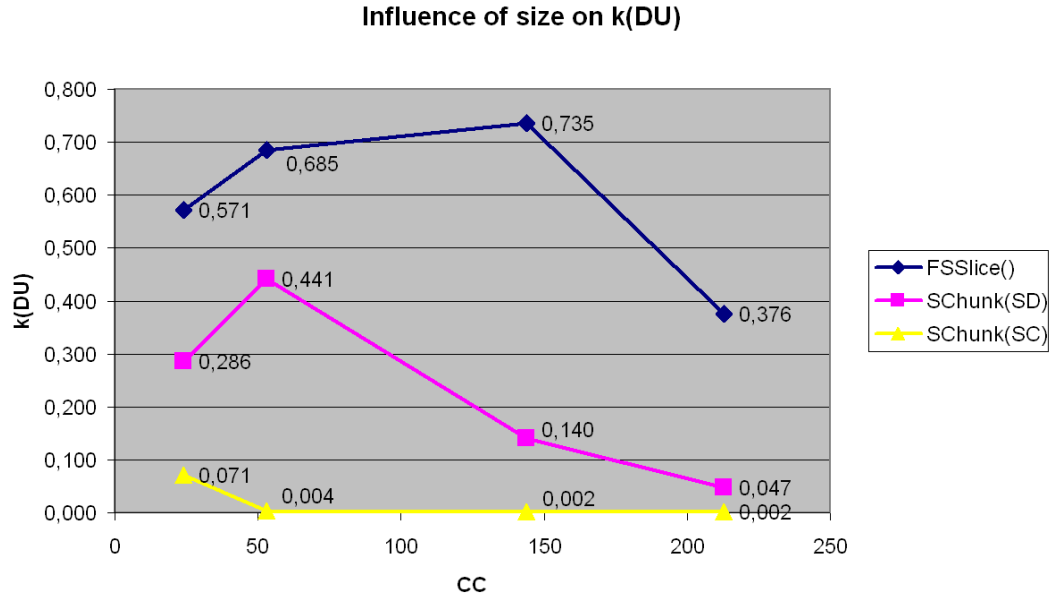


Fig. A.50: With the increasing size of specifications (described by CC), the approach reduces the number of DU to a much higher extent.

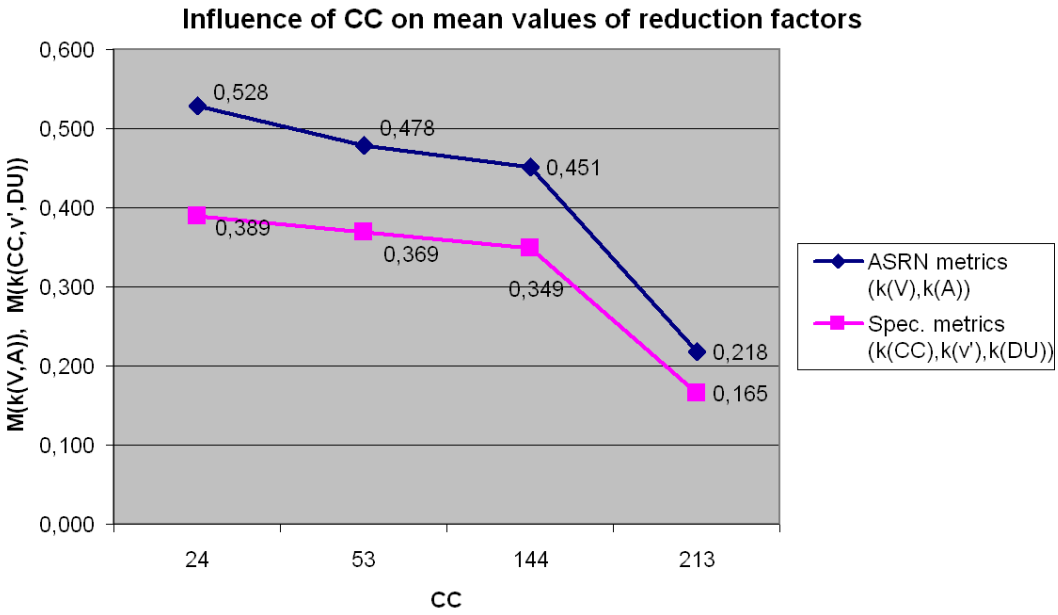


Fig. A.51: ASRN characteristics (V, A) and characteristics of the specification (CC, v', DU). With the increasing size of the specification (described by CC), the approach reduces the characteristics to a much higher extent.

A.5 Points of Interest and Response Variables

This section presents statistical data concerning the response variables (mean value, deviation, variance, minimum and maximum value).

Additionally it provides a complete listing of points of interest (POIs). The format is as follows: First there is a list of POIs which contains the name of the specification, the unique vertex ID, the number of the corresponding SRN block and the literals belonging to that prime. The list of POIs is followed by the full listing of primes of the corresponding specification.

Finally, the full list of all response variables is provided. Every variable is labelled with the corresponding specification name (*Specs.*), the abstraction criterion (*Deps.*) and the unique vertex ID (V-*ID*). The response variables are: $V \ldots$ number of vertices, $A \ldots$ number of arcs, $S/E \ldots$ number of start and end vertices (which represents the number of SRN blocks), $P \ldots$ number of prime vertices, $CC \ldots$ the conceptual complexity, $v'(l) \ldots$ lower bound of extended cyclomatic complexity, $v'(u) \ldots$ upper bound of extended cyclomatic complexity and $DU \ldots$ the definition/use count measure.

Spec.(Dep)	N/Stat.	V	A	k(V)	k(A)	f(k)	CC	v'(l)	v'(u)	DU	k(CC)	k(v'(l))	k(v'(u))	k(DU)
BB Total	**7 POI**	**72**	**267**				**24**	**4**	**10**	**4**				
BB (SCD)	Mean	47,143	148,143	0,655	0,555	0,800	13,000	2,714	4,857	2,286	0,542	0,679	0,486	0,571
	Dev.	13,607	69,261	0,189	0,259	0,201	6,856	1,604	3,761	2,138	0,286	0,401	0,376	0,535
	Var.	158,694	4111,837	0,031	0,058	0,035	40,286	2,204	12,122	3,918	0,070	0,138	0,121	0,245
	Min.	26,000	43,000	0,361	0,161	0,446	4,000	1,000	1,000	0,000	0,167	0,250	0,100	0,000
	Max.	66,000	232,000	0,917	0,869	0,949	22,000	4,000	10,000	4,000	0,917	1,000	1,000	1,000
BB (SD)	Mean	40,714	114,571	0,565	0,429	0,729	9,286	1,143	1,286	1,143	0,387	0,286	0,129	0,286
	Dev.	9,621	46,303	0,134	0,173	0,149	4,271	0,378	0,756	1,574	0,178	0,094	0,076	0,393
	Var.	79,347	1837,673	0,015	0,026	0,019	15,633	0,122	0,490	2,122	0,027	0,008	0,005	0,133
	Min.	24,000	43,000	0,333	0,161	0,483	4,000	1,000	1,000	0,000	0,167	0,250	0,100	0,000
	Max.	50,000	168,000	0,694	0,629	0,906	15,000	2,000	3,000	4,000	0,625	0,500	0,300	1,000
BB (SC)	Mean	37,857	117,000	0,526	0,438	0,814	10,429	2,286	2,286	0,286	0,435	0,571	0,229	0,071
	Dev.	18,828	65,271	0,261	0,244	0,086	6,051	1,604	1,604	0,488	0,252	0,401	0,160	0,122
	Var.	303,837	3651,714	0,059	0,051	0,006	31,388	2,204	2,204	0,204	0,054	0,138	0,022	0,013
	Min.	14,000	43,000	0,194	0,161	0,678	4,000	1,000	1,000	0,000	0,167	0,250	0,100	0,000
	Max.	63,000	202,000	0,875	0,757	0,908	19,000	4,000	4,000	1,000	0,792	1,000	0,400	0,250
Petrol Total	**21 POI**	**134**	**674**				**53**	**10**	**28**	**131**				
Petrol (SCD)	Mean	95,000	445,429	0,709	0,661	0,835	36,381	8,286	22,857	89,762	0,686	0,829	0,816	0,685
	Dev.	28,157	207,255	0,210	0,308	0,346	17,250	3,621	10,864	44,651	0,325	0,362	0,388	0,341
	Var.	755,048	40909,293	0,042	0,090	0,114	283,379	12,490	112,408	1898,753	0,101	0,125	0,143	0,111
	Min.	32,000	5,000	0,239	0,007	0,031	1,000	1,000	1,000	0,000	0,019	0,100	0,036	0,000
	Max.	116,000	579,000	0,866	0,859	1,015	47,000	10,000	28,000	115,000	0,887	1,000	1,000	0,878
Petrol (SD)	Mean	84,619	343,667	0,631	0,510	0,730	29,476	1,381	2,143	57,762	0,556	0,138	0,077	0,441
	Dev.	23,472	157,336	0,175	0,233	0,298	13,819	0,498	1,493	28,804	0,261	0,050	0,053	0,220
	Var.	524,712	23575,746	0,029	0,052	0,084	181,868	0,236	2,122	790,181	0,065	0,002	0,003	0,046
	Min.	30,000	5,000	0,224	0,007	0,031	1,000	1,000	1,000	0,000	0,019	0,100	0,036	0,000
	Max.	100,000	441,000	0,746	0,654	0,916	38,000	2,000	4,000	75,000	0,717	0,200	0,143	0,573
Petrol (SC)	Mean	32,381	79,429	0,242	0,118	0,479	7,810	2,000	2,000	0,524	0,147	0,200	0,071	0,004
	Dev.	8,997	38,966	0,067	0,058	0,188	4,457	1,643	1,643	0,928	0,084	0,164	0,059	0,007
	Var.	77,093	1446,054	0,004	0,003	0,034	18,916	2,571	2,571	0,821	0,007	0,026	0,003	0,000
	Min.	14,000	5,000	0,104	0,007	0,040	1,000	1,000	1,000	0,000	0,019	0,100	0,036	0,000
	Max.	45,000	136,000	0,336	0,202	0,683	14,000	5,000	5,000	3,000	0,264	0,500	0,179	0,023

Fig. A.52: Statistics concerning the 7 (21) points of interest regarding to the *BB* (*Petrol*) specification and three types of abstraction.

Spec.(Dep)	N/Stat.	V	A	k(V)	k(A)	f(k)	CC	v'(l)	v'(u)	DU	k(CC)	k(v'(l))	k(v'(u))	k(DU)
Elevator Total	**102 POI**	**349**	**3668**				**144**	**32**	**1069**	**1212**				
Elevator (SCD)	Mean	289,059	2759,765	0,828	0,752	0,863	108,363	29,265	831,647	890,853	0,753	0,915	0,778	0,735
	Dev.	52,461	845,088	0,150	0,230	0,253	32,569	8,836	264,320	279,279	0,226	0,276	0,247	0,230
	Var.	2725,134	707172,709	0,022	0,053	0,063	1050,310	77,312	69180,228	77232,125	0,051	0,076	0,061	0,053
	Min.	116,000	50,000	0,332	0,014	0,041	4,000	1,000	1,000	0,000	0,028	0,031	0,001	0,000
	Max.	322,000	3341,000	0,923	0,911	0,992	131,000	32,000	1069,000	1051,000	0,910	1,000	1,000	0,867
Elevator (SD)	Mean	191,167	710,216	0,548	0,194	0,326	42,157	1,245	5,069	169,951	0,293	0,039	0,005	0,140
	Dev.	42,187	365,937	0,121	0,100	0,140	35,290	12,313	323,400	339,387	0,257	0,397	0,323	0,300
	Var.	1762,335	132597,012	0,014	0,010	0,019	466,897	0,185	69,868	22683,439	0,023	0,000	0,000	0,015
	Min.	89,000	50,000	0,255	0,014	0,041	4,000	1,000	1,000	0,000	0,028	0,031	0,001	0,000
	Max.	248,000	1317,000	0,711	0,359	0,597	76,000	2,000	42,000	483,000	0,528	0,063	0,039	0,399
Elevator (SC)	Mean	110,000	261,284	0,315	0,071	0,219	23,569	11,471	11,471	2,971	0,164	0,358	0,011	0,002
	Dev.	50,378	182,635	0,144	0,050	0,089	17,711	10,686	10,686	4,329	0,123	0,334	0,010	0,004
	Var.	2513,039	33028,458	0,021	0,002	0,008	310,598	113,073	113,073	18,558	0,015	0,110	0,000	0,000
	Min.	8,000	20,000	0,023	0,005	0,041	3,000	1,000	1,000	0,000	0,021	0,031	0,001	0,000
	Max.	171,000	521,000	0,490	0,142	0,324	47,000	25,000	25,000	14,000	0,326	0,781	0,023	0,012
WM Total	**92 POI**	**520**	**2633**				**213**	**39**	**544**	**215**				
WM (SCD)	Mean	236,087	958,087	0,454	0,364	0,638	76,413	20,620	167,217	80,815	0,359	0,529	0,307	0,376
	Dev.	131,113	682,892	0,252	0,259	0,336	54,056	14,742	123,169	63,543	0,254	0,378	0,226	0,296
	Var.	17003,645	461271,927	0,063	0,067	0,112	2890,264	214,953	15005,605	3993,781	0,064	0,141	0,051	0,086
	Min.	16,000	9,000	0,031	0,003	0,020	1,000	1,000	1,000	0,000	0,005	0,026	0,002	0,000
	Max.	400,000	1746,000	0,769	0,663	0,913	140,000	39,000	286,000	170,000	0,657	1,000	0,526	0,791
WM (SD)	Mean	91,739	159,283	0,176	0,060	0,323	13,076	1,196	1,565	10,022	0,061	0,031	0,003	0,047
	Dev.	37,282	116,593	0,072	0,044	0,166	9,428	0,399	1,718	11,927	0,044	0,010	0,003	0,055
	Var.	1374,867	13446,181	0,005	0,002	0,027	87,918	0,157	2,920	140,717	0,002	0,000	0,000	0,003
	Min.	16,000	9,000	0,031	0,003	0,021	1,000	1,000	1,000	0,000	0,005	0,026	0,002	0,000
	Max.	172,000	407,000	0,331	0,155	0,686	35,000	2,000	10,000	42,000	0,164	0,051	0,018	0,195
WM (SC)	Mean	96,217	183,696	0,185	0,070	0,319	18,707	6,554	6,554	0,413	0,088	0,168	0,012	0,002
	Dev.	58,541	195,991	0,113	0,074	0,190	20,734	8,249	8,249	1,150	0,097	0,212	0,015	0,005
	Var.	3389,801	37994,994	0,013	0,005	0,036	425,207	67,312	67,312	1,308	0,009	0,044	0,000	0,000
	Min.	15,000	9,000	0,029	0,003	0,023	1,000	1,000	1,000	0,000	0,005	0,026	0,002	0,000
	Max.	199,000	542,000	0,383	0,206	0,665	56,000	20,000	20,000	7,000	0,263	0,513	0,037	0,033

Fig. A.53: Statistics concerning the 92 (102) points of interest regarding to the *WM* (*Elevator*) specification and three types of abstraction.

Fig. A.54: Detailed list of all points of interest for the *BB*-, *Petrol*-, *Elevator*- and *WM*-specification. The figure contains the list of points of interest and the annotated listing of the specification.

```
34/4/2   \end{schema}

54/5/1   \begin{schema}{ArriveAndWait}
56/5/3   \Delta Tankstelle
57/5/5   \\
58/5/3   fz? : VH
59/5/5   \\
60/5/3   z? : GS
61/5/4   \where
62/5/3   z? \in petrol
63/5/5   \\
64/5/3   \sharp (\ran operation) = \sharp petrol
65/5/5   \\
66/5/3   fz? \notin \ran operation
67/5/5   \\
68/5/3   \forall GS : GS | GS \in petrol @ fz? \notin
         \ran (waiting \,GS)
69/5/5   \\
70/5/3   operation' = operation
71/5/5   \\
72/5/3   waiting' = waiting \oplus \{ z? \mapsto
         (waiting \,z?) \cat \lseq fz? \rseq \}
73/5/5   \\
74/5/3   petrol' = petrol
55/5/2   \end{schema}

75/6/1   \begin{zed}
76/6/2   Arrival == ArriveAndFuel \lor ArriveAndWait
77/6/3   \end{zed}

78/7/1   \begin{schema}{LeaveEmptyQueue}
80/7/3   \Delta PetrolStation
81/7/5   \\
82/7/3   fz? : VH
83/7/5   \\
84/7/3   z : GS
85/7/4   \where
86/7/3   \exists_1 z : GS | operation \, z = fz?
         \land waiting \, z = \emptyseq @ operation' =
         operation \setminus \{ z \mapsto fz? \} \land
         waiting' = waiting \land petrol' = petrol
79/7/2   \end{schema}

87/8/1   \begin{schema}{LeaveNonEmptyQueue}
89/8/3   \Delta PetrolStation
90/8/5   \\
91/8/3   fz? : VH

92/8/5   \\
93/8/3   z : GS
94/8/4   \where
95/8/3   \exists_1 z : GS | operation \, z = fz?
         \land waiting \, z \neq \emptyseq @ operation' =
         operation \oplus \{ z \mapsto \head (waiting \, z) \}
         \land waiting' = waiting \oplus \{ z \mapsto \tail
         (waiting \, z) \} \land petrol' = petrol
88/8/2   \end{schema}

96/9/1   \begin{schema}{LeaveWithoutFuel}
98/9/3   \Delta PetrolStation
99/9/5   \\

100/9/3  fz? : VH
101/9/4  \where
102/9/3  \exists z : GS; i : \num | z \in
         petrol \land  (i \mapsto fz?) \in waiting \,
         z @ waiting' = waiting \oplus \{ z \mapsto
         \squash ( (waiting \,z) \setminus \{ i
         \mapsto fz? \} \,) \,\} \land operation' =
         operation \land petrol' = petrol
97/9/2   \end{schema}

103/10/1 \begin{zed}
105/10/3 Leave
106/10/4 == \\
107/11/3 LeaveEmptyQueue
108/11/5 \lor
109/11/3 LeaveNonEmptyQueue
112/12/5 \lor
113/12/3 LeaveWithoutFuel
104/10/2 \end{zed}
118/13/1 \begin{zed}
120/13/3 PetrolStation
121/13/4 ==
122/14/3 InitPetrolStation
123/15/5 \land
124/14/7 ( \,
125/14/3 Arrive
126/14/5 \lor
127/14/3 Leave \,
130/14/8 )
119/13/2 \end{zed}

***Elevator-specification***

**Points of interest (Name,Vertex,SRN-Block):**

Elevator, 21, 2, CurrentFloor \leq MaxFloor
Elevator, 23, 2, max ( Requests ) \leq
MaxFloor
Elevator, 25, 2, max ( UpCalls ) \leq
MaxFloor
Elevator, 27, 2, max ( DownCalls ) \leq
MaxFloor
Elevator, 32, 3, CurrentFloor = 1
Elevator, 34, 3, Requests = \emptyset
Elevator, 36, 3, UpCalls = \emptyset
Elevator, 38, 3, DownCalls = \emptyset
Elevator, 40, 3, Dir = up
Elevator, 42, 3, Door = open
Elevator, 47, 4, Requests \cup UpCalls \cup
DownCalls = \emptyset
Elevator, 54, 5, Floor? \leq MaxFloor
Elevator, 56, 5, CurrentFloor' =
CurrentFloor
Elevator, 58, 5, Requests' = Requests \cup
\{Floor?\}
Elevator, 60, 5, UpCalls' = UpCalls
Elevator, 62, 5, DownCalls' = DownCalls
Elevator, 64, 5, Dir' = Dir
Elevator, 66, 5, Door' = Door
Elevator, 75, 6, Floor? \leq MaxFloor
Elevator, 77, 6, CurrentFloor' = CurrentFloor
Elevator, 79, 6, (CallDir? = down) \implies
(UpCalls' = UpCalls) \land ( DownCalls' =
DownCalls \cup \{ Floor? \} )
Elevator, 81, 6, (CallDir? = up) \implies
(DownCalls' = DownCalls) \land (UpCalls' = UpCalls
\cup \{Floor?\})
Elevator, 83, 6, Requests' = Requests
Elevator, 85, 6, Dir' = Dir
Elevator, 87, 6, Door' = Door
Elevator, 102, 9, Dir = up
Elevator, 104, 9, Requests \cup UpCalls \neq
\emptyset
Elevator, 106, 9, CurrentFloor \leq max ( Requests
\cup UpCalls)
Elevator, 108, 9, CurrentFloor' = min  \{ x :
\nat_1 | x \in (Requests \cup UpCalls) \land x >
CurrentFloor \}
Elevator, 111, 9, Requests' = Requests \setminus
\{ CurrentFloor' \}
Elevator, 113, 9, UpCalls' = UpCalls
Elevator, 115, 9, DownCalls' = DownCalls \setminus
\{ CurrentFloor' \}
Elevator, 117, 9, Dir' = up
Elevator, 119, 9, Door' = Door
Elevator, 124, 10, Dir = down
Elevator, 126, 10, Requests \cup DownCalls \neq
\emptyset
Elevator, 128, 10, CurrentFloor \leq min (
Requests \cup DownCalls)
Elevator, 130, 10, CurrentFloor' = max  \{ x :
\nat_1 | x \in (Requests \cup DownCalls) \land x <
CurrentFloor \}
Elevator, 133, 10, Requests' = Requests \setminus
\{ CurrentFloor' \}
Elevator, 135, 10, UpCalls' = UpCalls \setminus \{
CurrentFloor' \}
Elevator, 137, 10, DownCalls' = DownCalls
Elevator, 139, 10, Dir' = down
Elevator, 141, 10, Door' = Door
Elevator, 146, 11, Dir = up
Elevator, 148, 11, (Requests \cup UpCalls \neq
\emptyset \land CurrentFloor > max  ( Requests
\cup UpCalls)) \\ \t2\lor Requests \cup UpCalls =
\emptyset
Elevator, 150, 11, Requests \cup DownCalls \neq
\emptyset
Elevator, 152, 11, CurrentFloor' = max  ( Requests
\cup DownCalls )
Elevator, 154, 11, Requests' = Requests \setminus
\{ CurrentFloor' \}
Elevator, 156, 11, UpCalls' = UpCalls
Elevator, 158, 11, DownCalls' = DownCalls
\setminus \{ CurrentFloor' \}
Elevator, 160, 11, Dir' = down
Elevator, 162, 11, Door' = Door
Elevator, 167, 12, Dir = down
Elevator, 169, 12, (Requests \cup DownCalls \neq
\emptyset \land CurrentFloor < min (Requests \cup
DownCalls ) ) \\ \t2\lor Requests \cup DownCalls =
\emptyset
```

Fig. A.55: Detailed list of all points of interest for the *BB*-, *Petrol*-, *Elevator*- and *WM*-specification (part 2 of 8). The figure contains the list of points of interest and the annotated listing of the specification.

```
Elevator, 171, 12, Requests \cup UpCalls \neq
\emptyset
Elevator, 173, 12, CurrentFloor' = min ( Requests
\cup UpCalls' )
Elevator, 175, 12, Requests' = Requests \setminus \{
CurrentFloor' \}
Elevator, 177, 12, UpCalls' = UpCalls \setminus \{
CurrentFloor' \}
Elevator, 179, 12, DownCalls' = DownCalls
Elevator, 181, 12, Dir' = up
Elevator, 183, 12, Door' = Door
Elevator, 188, 13, Dir = up
Elevator, 190, 13, UpCalls \neq \emptyset
Elevator, 192, 13, CurrentFloor > max ( UpCalls )
Elevator, 194, 13, DownCalls \cup Requests =
\emptyset
Elevator, 196, 13, CurrentFloor' = min ( UpCalls )
Elevator, 198, 13, Requests' = Requests
Elevator, 200, 13, UpCalls' = UpCalls \setminus
Elevator, 202, 13, \{ CurrentFloor' \}
Elevator, 204, 13, DownCalls' = DownCalls
Elevator, 206, 13, Dir' = up
Elevator, 208, 13, Door' = Door
Elevator, 213, 14, Dir = down
Elevator, 215, 14, DownCalls \neq \emptyset
Elevator, 217, 14, CurrentFloor < min ( DownCalls )
Elevator, 219, 14, UpCalls \cup Requests = \emptyset
Elevator, 221, 14, CurrentFloor' = max \ DownCalls
Elevator, 223, 14, Requests' = Requests
Elevator, 225, 14, UpCalls' = UpCalls
Elevator, 227, 14, DownCalls' = DownCalls \setminus
\{ CurrentFloor' \}
Elevator, 229, 14, Dir' = down
Elevator, 231, 14, Door' = Door
Elevator, 266, 21, CurrentFloor' = CurrentFloor
Elevator, 268, 21, Requests' = Requests
Elevator, 270, 21, UpCalls' = UpCalls
Elevator, 272, 21, DownCalls' = DownCalls
Elevator, 274, 21, Dir' = Dir
Elevator, 276, 21, Door'= open
Elevator, 281, 22, Requests \cup DownCalls \cup
UpCalls \neq \emptyset
Elevator, 283, 22, CurrentFloor' = CurrentFloor
Elevator, 285, 22, Requests' = Requests
Elevator, 287, 22, UpCalls' = UpCalls
Elevator, 289, 22, DownCalls' = DownCalls
Elevator, 291, 22, Dir' = Dir
Elevator, 293, 22, Door'= closed
Elevator, 300, 24, PassengerEvent
Elevator, 304, 25, Move
Elevator, 310, 26, PassengerEvent
Elevator, 325, 29, PassengerEvent
Elevator, 335, 31, MoveCycle
Elevator, 342, 32, PassengerEvent
Elevator, 344, 32, ElevatorCycle

Listing (Vertex/SRN-Block,Type):

2/1/1    \begin{zed}
4/1/3    MaxFloor == 10 \\
5/1/3    Direction ::= up | down \\

6/1/3    DoorState ::= open | closed
3/1/2    \end{zed}

7/2/1    \begin{schema}{Elevator}
9/2/3    CurrentFloor : \nat_1
10/2/5   \\
11/2/3   Requests : \power \nat_1
12/2/5   \\
13/2/3   UpCalls : \power \nat_1
14/2/5   \\
15/2/3   DownCalls : \power \nat_1
16/2/5   \\
17/2/3   Dir : Direction
18/2/5   \\
19/2/3   Door : DoorState
20/2/4   \where
21/2/3   CurrentFloor \leq MaxFloor
22/2/5   \\
23/2/3   max ( Requests ) \leq MaxFloor
24/2/5   \\
25/2/3   max ( UpCalls ) \leq MaxFloor
26/2/5   \\
27/2/3   max ( DownCalls ) \leq MaxFloor
8/2/2    \end{schema}

28/3/1   \begin{schema}{InitElevator}
30/3/3   Elevator
31/3/4   \where
32/3/3   CurrentFloor = 1
33/3/5   \\
34/3/3   Requests = \emptyset
35/3/5   \\
36/3/3   UpCalls = \emptyset
37/3/5   \\
38/3/3   DownCalls = \emptyset
39/3/5   \\
40/3/3   Dir = up
41/3/5   \\
42/3/3   Door = open
29/3/2   \end{schema}

43/4/1   \begin{schema}{NoRequestsOrCalls}
45/4/3   \Xi Elevator
46/4/4   \where
47/4/3   Requests \cup UpCalls \cup
DownCalls = \emptyset
44/4/2   \end{schema}

48/5/1   \begin{schema}{ElevatorButtonEvent}
50/5/3   \Delta Elevator
51/5/5   \\
52/5/3   Floor? : \nat_1
53/5/4   \where
54/5/3   Floor? \leq MaxFloor
55/5/5   \\
56/5/3   CurrentFloor' = CurrentFloor
57/5/5   \\
58/5/3   Requests' = Requests \cup
\{Floor?\}
59/5/5   \\
60/5/3   UpCalls' = UpCalls

61/5/5   \\
62/5/3   DownCalls' = DownCalls
63/5/5   \\
64/5/3   Dir' = Dir
65/5/5   \\
66/5/3   Door' = Door
49/5/2   \end{schema}

67/6/1   \begin{schema}{FloorButtonEvent}
69/6/3   \Delta Elevator
70/6/5   \\
71/6/3   Floor? : \nat_1
72/6/5   \\
73/6/3   CallDir? : Direction
74/6/4   \where
75/6/3   Floor? \leq MaxFloor
76/6/5   \\
77/6/3   CurrentFloor' = CurrentFloor
78/6/5   \\
79/6/3   (CallDir? = down) \implies (UpCalls' =
UpCalls) \land ( DownCalls' = DownCalls \cup \{
Floor? \} )
80/6/5   \\
81/6/3   (CallDir? = up) \implies (DownCalls' =
DownCalls) \land (UpCalls' = UpCalls \cup
\{Floor?\})
82/6/5   \\
83/6/3   Requests' = Requests
84/6/5   \\
85/6/3   Dir' = Dir
86/6/5   \\
87/6/3   Door' = Door
68/6/2   \end{schema}

88/7/1   \begin{zed}
90/7/3   PassengerEvent
91/7/4   ==
92/8/3   ElevatorButtonEvent
93/8/5   \lor
94/8/3   FloorButtonEvent
89/7/2   \end{zed}

98/9/1   \begin{schema}{BasicMoveUp}
100/9/3  \Delta Elevator
101/9/4  \where
102/9/3  Dir = up
103/9/5  \\
104/9/3  Requests \cup UpCalls \neq \emptyset
105/9/5  \\
106/9/3  CurrentFloor \leq max ( Requests \cup
UpCalls)
107/9/5  \\
108/9/3  CurrentFloor' = min \{ x : \nat_1 | x
\in (Requests \cup UpCalls) \land x > CurrentFloor
\}
110/9/5  \\
111/9/3  Requests' = Requests \setminus \{
CurrentFloor' \}
112/9/5  \\
113/9/3  UpCalls' = UpCalls
114/9/5  \\
```

Fig. A.56: Detailed list of all points of interest for the *BB*-, *Petrol*-, *Elevator*- and *WM*-specification (part 3 of 8). The figure contains the list of points of interest and the annotated listing of the specification.

```
115/9/3  DownCalls' = DownCalls \setminus \{
CurrentFloor' \}
116/9/5  \\
117/9/3  Dir' = up
118/9/5  \\
119/9/3  Door' = Door
99/9/2   \end{schema}

120/10/1 \begin{schema}[BasicMoveDown]
122/10/3 \Delta Elevator
123/10/4 \where
124/10/3 Dir = down
125/10/5 \\
126/10/3 Requests \cup DownCalls \neq \emptyset
127/10/5 \\
128/10/3 CurrentFloor \leq min ( Requests \cup
DownCalls)
129/10/5 \\
130/10/3 CurrentFloor' = max  \{ x : \nat_1 | x \in
(Requests \cup DownCalls) \land x < CurrentFloor \}
132/10/5 \\
133/10/3 Requests' = Requests \setminus \{
CurrentFloor' \}
134/10/5 \\
135/10/3 UpCalls' = UpCalls \setminus \{
CurrentFloor' \}
136/10/5 \\
137/10/3 DownCalls' = DownCalls
138/10/5 \\
139/10/3 Dir' = down
140/10/5 \\
141/10/3 Door' = Door
121/10/2 \end{schema}

142/11/1 \begin{schema}[ChangeUpToDown]
144/11/3 \Delta Elevator
145/11/4 \where
146/11/3 Dir = up
147/11/5 \\
148/11/3 (Requests \cup UpCalls \neq \emptyset \land
CurrentFloor > max  ( Requests \cup UpCalls)) \\
\t2\lor Requests \cup UpCalls = \emptyset
149/11/5 \\
150/11/3 Requests \cup DownCalls \neq \emptyset
151/11/5 \\
152/11/3 CurrentFloor' = max  ( Requests \cup
DownCalls )
153/11/5 \\
154/11/3 Requests' = Requests \setminus \{
CurrentFloor' \}
155/11/5 \\
156/11/3 UpCalls' = UpCalls
157/11/5 \\
158/11/3 DownCalls' = DownCalls \setminus \{
CurrentFloor' \}
159/11/5 \\
160/11/3 Dir' = down
161/11/5 \\
162/11/3 Door' = Door
143/11/2 \end{schema}

163/12/1 \begin{schema}[ChangeDownToUp]
165/12/3 \Delta Elevator
166/12/4 \where
167/12/3 Dir = down
168/12/5 \\
169/12/3 (Requests \cup DownCalls \neq
\emptyset \land CurrentFloor < min (Requests
\cup DownCalls ) ) \\ \t2\lor Requests \cup
DownCalls = \emptyset
170/12/5 \\
171/12/3 Requests \cup UpCalls \neq
\emptyset
172/12/5 \\
173/12/3 CurrentFloor' = min ( Requests
\cup UpCalls )
174/12/5 \\
175/12/3 Requests' = Requests \setminus \{
CurrentFloor' \}
176/12/5 \\
177/12/3 UpCalls' = UpCalls \setminus \{
CurrentFloor' \}
178/12/5 \\
179/12/3 DownCalls' = DownCalls
180/12/5 \\
181/12/3 Dir' = up
182/12/5 \\
183/12/3 Door' = Door
164/12/2 \end{schema}

184/13/1 \begin{schema}[RestartMovingUp]
186/13/3 \Delta Elevator
187/13/4 \where
188/13/3 Dir = up
189/13/5 \\
190/13/3 UpCalls \neq \emptyset
191/13/5 \\
192/13/3 CurrentFloor > max ( UpCalls )
193/13/5 \\
194/13/3 DownCalls \cup Requests =
\emptyset
195/13/5 \\
196/13/3 CurrentFloor' = min ( UpCalls )
197/13/5 \\
198/13/3 Requests' = Requests
199/13/5 \\
200/13/3 UpCalls' = UpCalls \setminus
201/13/5 \\
202/13/3 \{ CurrentFloor' \}
203/13/5 \\
204/13/3 DownCalls' = DownCalls
205/13/5 \\
206/13/3 Dir' = up
207/13/5 \\
208/13/3 Door' = Door
185/13/2 \end{schema}

209/14/1 \begin{schema}[RestartMovingDown]
211/14/3 \Delta Elevator
212/14/4 \where
213/14/3 Dir = down
214/14/5 \\
215/14/3 DownCalls \neq \emptyset
216/14/5 \\
217/14/3 CurrentFloor < min ( DownCalls )
218/14/5 \\
219/14/3 UpCalls \cup Requests = \emptyset
220/14/5 \\
221/14/3 CurrentFloor' = max \ DownCalls
222/14/5 \\
223/14/3 Requests' = Requests
224/14/5 \\
225/14/3 UpCalls' = UpCalls
226/14/5 \\
227/14/3 DownCalls' = DownCalls \setminus \{
CurrentFloor' \}
228/14/5 \\
229/14/3 Dir' = down
230/14/5 \\
231/14/3 Door' = Door
210/14/2 \end{schema}

232/15/1 \begin{zed}
234/15/3 Move
235/15/4 ==
236/16/3 BasicMoveUp
237/16/5 \lor
238/16/3 BasicMoveDown
241/17/5 \lor
242/17/3 ChangeUpToDown
246/18/5 \lor
247/18/3 ChangeDownToUp
251/19/5 \lor
252/19/3 RestartMovingUp
256/20/5 \lor
257/20/3 RestartMovingDown
233/15/2 \end{zed}

262/21/1 \begin{schema}[OpenDoor]
264/21/3 \Delta Elevator
265/21/4 \where
266/21/3 CurrentFloor' = CurrentFloor
267/21/5 \\
268/21/3 Requests' = Requests
269/21/5 \\
270/21/3 UpCalls' = UpCalls
271/21/5 \\
272/21/3 DownCalls' = DownCalls
273/21/5 \\
274/21/3 Dir' = Dir
275/21/5 \\
276/21/3 Door'= open
263/21/2 \end{schema}

277/22/1 \begin{schema}[CloseDoor]
279/22/3 \Delta Elevator
280/22/4 \where
281/22/3 Requests \cup DownCalls \cup UpCalls
\neq \emptyset
282/22/5 \\
283/22/3 CurrentFloor' = CurrentFloor
284/22/5 \\
285/22/3 Requests' = Requests
```

Fig. A.57: Detailed list of all points of interest for the *BB*-, *Petrol*-, *Elevator*- and *WM*-specification (part 4 of 8). The figure contains the list of points of interest and the annotated listing of the specification.

```
286/22/5  \\
287/22/3  UpCalls' = UpCalls
288/22/5  \\
289/22/3  DownCalls' = DownCalls
290/22/5  \\
291/22/3  Dir' = Dir
292/22/5  \\
293/22/3  Door'= closed
278/22/2  \end{schema}

294/23/1  \begin{zed}
296/23/3  MoveCycle
297/23/4  ==
298/24/3  CloseDoor
299/24/5  \semi
300/24/3  PassengerEvent
303/25/5  \semi
304/25/3  Move
308/27/5  \semi
309/26/7  (
310/26/3  PassengerEvent
311/26/8  )
318/28/5  \semi
319/28/3  OpenDoor
323/30/5  \semi
324/29/7  (
325/29/3  PassengerEvent
326/29/8  )
333/23/3  \\ ElevatorCycle
334/23/4  ==
335/31/3  MoveCycle
336/31/5  \lor
337/31/3  NoRequestsOrCalls
340/23/3  \\ FunctioningElevator
341/23/4  ==
342/32/3  PassengerEvent
343/32/5  \semi
344/32/3  ElevatorCycle
295/23/2  \end{zed}

WM-specification

Points of interest (Name,Vertex,SRN-Block):

WM, 30, 5, pix_1 = (x_1, y_1)
WM, 32, 5, pix_2 = (x_2, y_2)
WM, 83, 17, \langle header, body \rangle \partition
map
WM, 85, 17, area = \dom map
WM, 97, 19, (first \ xylimits) \leq (second \
xylimits)
WM, 124, 24, Undefined \notin windows
WM, 126, 24, windows = \dom contents
WM, 128, 24, current \in windows \cup \{ Undefined \}
WM, 141, 25, maps = contents \comp
WM, 142, 25, (\lambda Info @ map)
WM, 144, 25, areas = contents \comp
WM, 145, 25, (\lambda Info @ area)
WM, 147, 25, \dis}oint areas

WM, 149, 25, \bigcup (\ran areas) \subseteq
\dom background
WM, 151, 25, screen = background \oplus
\bigcup (\ran maps)
WM, 156, 26, MaxWindows = 20
WM, 161, 27, \sharp windows \leq MaxWindows
WM, 172, 29, \forall w,w' : WINDOWS | w' =
adjust(w) @ \\ \t2\sharp w' = \sharp w \land
\\ \t2w' \comp
WM, 173, 29, (\lambda Info @ \Theta Control)
= w \comp (\lambda Info @ \Theta Control)
WM, 176, 30, WM'
WM, 178, 30, windows' = \emptyset
WM, 180, 30, current' = undefined
WM, 185, 31, \Delta Info
WM, 187, 31, current \in windows
WM, 189, 31, current' = current
WM, 191, 31, \Theta Info = contents
(current)
WM, 193, 31, contents' = adjust ( contents
\bigoplus \{current \mapsto \Theta Info' \}
)
WM, 200, 32, Info
WM, 202, 32, \sharp windows < MaxWindows
WM, 204, 32, w! \notin windows \cup \{
Undefined \}
WM, 206, 32, current' = w!
WM, 208, 32, control = Expose
WM, 210, 32, contents' = adjust ( contents
\cup \{ w! \mapsto \Theta Info \} )
WM, 215, 33, current \in windows
WM, 217, 33, current' = Undefined
WM, 219, 33, contents' = adjust (
\{current\} \ndres contents )
WM, 228, 34, map' = map
WM, 230, 34, title' = title
WM, 232, 34, control' = control
WM, 234, 34, xylimits' = (minxy?, maxxy?)
WM, 245, 35, \Theta Info' = \Theta Info
WM, 247, 35, \dom body = xy1 \dots xy2
WM, 249, 35, wh! = xy2 - xy1
WM, 254, 36, map' = map
WM, 256, 36, title' = title
WM, 258, 36, control' = Hide
WM, 260, 36, xylimits' = xylimits
WM, 265, 37, map' = map
WM, 267, 37, title' = title
WM, 269, 37, control' = Expose
WM, 271, 37, xylimits' = xylimits
WM, 278, 38, w? \in windows
WM, 280, 38, current' = w?
WM, 282, 38, contents' = contents
WM, 289, 39, map' = map
WM, 291, 39, title' = s?
WM, 293, 39, control' = control
WM, 295, 39, xylimits' = xylimits
WM, 300, 40, header' = header
WM, 302, 40, body' = setval \ White \ body
WM, 304, 40, title' = title
WM, 306, 40, control' = control
WM, 308, 40, xylimits' = xylimits

WM, 316, 42, Null \notin windows
WM, 323, 43, w! \neq Null
WM, 330, 44, \sharp windows \geq MaxWindows
WM, 332, 44, w! = Null
WM, 354, 48, current = Undefined
WM, 411, 59, w? \notin windows
WM, 428, 62, \dom wms \subseteq hosts
WM, 430, 62, \dis}oint ( wms \comp
WM, 431, 62, ( \lambda WM @ windows ) )
WM, 434, 63, ITC'
WM, 436, 63, hosts' = \emptyset
WM, 448, 65, host? \notin hosts \cup EMPTYSTRING
WM, 450, 65, hosts' = hosts \cup \{ host? \}
WM, 452, 65, wms' = wms
WM, 459, 66, host? in hosts
WM, 461, 66, hosts' = hosts \setminus \{ host? \}
WM, 463, 66, wms' = wms
WM, 470, 67, hosts' = hosts
WM, 472, 67, localhosts \in hosts
WM, 479, 68, localhost \notin \dom wms
WM, 481, 68, wms' = wms \cup \{ localhost \mapsto
initwm \}
WM, 486, 69, localhost \in \dom wms
WM, 488, 69, wms' = \{ localhost \} \ndres wms
WM, 497, 70, \Theta WM = wms \ host
WM, 499, 70, \Theta WM' = wms' \ host
WM, 502, 71, NewWindow1
WM, 508, 71, host? = EMPTYSTRING \imp host =
localhost
WM, 510, 71, host? \neq EMPTYSTRING \imp host =
host?
WM, 515, 73, DeleteWindow1

Listing (Vertex/SRN-Block,Type):

2/1/1   \begin{axdef}
4/1/3   Xsize : \nat_1
5/1/5   \\
6/1/3   Ysize : \nat_1
3/1/2   \end{axdef}

7/2/1   \begin{zed}
9/2/3   Xrange == 0 \dots ( Xsize - 1 )
8/2/2   \end{zed}

10/3/1  \begin{zed}
12/3/3  Yrange == 0 \dots ( Ysize - 1 )
11/3/2  \end{zed}

13/4/1  \begin{zed}
15/4/3  Pixel == ( Xrange \cross Yrange )
14/4/2  \end{zed}

16/5/1  \begin{schema}{PixelPair}
18/5/3  pix_1 : Pixel
19/5/5  \\
20/5/3  pix_2 : Pixel
21/5/5  \\
22/5/3  x_1 : Xrange
23/5/5  \\
24/5/3  x_2 : Xrange
```

Fig. A.58: Detailed list of all points of interest for the *BB*-, *Petrol*-, *Elevator*- and *WM*-specification (part 5 of 8). The figure contains the list of points of interest and the annotated listing of the specification.

```
25/5/5   \\
26/5/5   y_1 : Yrange
27/5/5   \\
28/5/3   y_2 : Yrange
29/5/4   \where
30/5/3   pix_1 = (x_1, y_1)
31/5/5   \\
32/5/3   pix_2 = (x_2, y_2)
17/5/2   \end{schema}

33/6/1   \begin{axdef}
35/6/3   Zsize : \nat_1
34/6/2   \end{axdef}

36/7/1   \begin{zed}
38/7/3   ClearVal == 0
37/7/2   \end{zed}

39/8/1   \begin{zed}
41/8/3   SetVal == 1
40/8/2   \end{zed}

42/9/1   \begin{zed}
44/9/3   BitVal == \{ClearVal, SetVal \}
43/9/2   \end{zed}

45/10/1  \begin{zed}
47/10/3  Zrange == 0 \dots ( Zsize - 1 )
46/10/2  \end{zed}

48/11/1  \begin{zed}
50/11/3  Value == ( Zrange \fun BitVal )
49/11/2  \end{zed}

51/12/1  \begin{zed}
53/12/3  Black == ( \mu val : Value | \ran val = \{
ClearVal \} )
52/12/2  \end{zed}

55/13/1  \begin{zed}
57/13/3  White == ( \mu val : Value | \ran val = \{
SetVal \} )
56/13/2  \end{zed}

59/14/1  \begin{zed}
61/14/3  Pixmap == ( Pixel \pfun Value )
60/14/2  \end{zed}

62/15/1  \begin{axdef}
64/15/3  setval : V \fun P \pfun V \fun P \pfun V
65/15/5  \where
67/15/3  \forall v : V; p : ( P \pfun V) @ setval v p
= (\mu m : P \pfun V | (\dom m = \dom p \land \ran m
= \{ v \}))
63/15/2  \end{axdef}

68/16/1  \begin{zed}
70/16/4  [
71/16/3  String
72/16/4  ]
69/16/2  \end{zed}

73/17/1  \begin{schema}{Map}
75/17/3  header : Pixmap
76/17/5  \\
77/17/3  body  : Pixmap
78/17/5  \\
79/17/3  map : Pixmap
80/17/5  \\
81/17/3  area : \power Pixel
82/17/4  \where
83/17/3  \langle header, body \rangle
\partition map
84/17/5  \\
85/17/3  area = \dom map
74/17/2  \end{schema}

86/18/1  \begin{zed}
88/18/3  HideExpose ::= Hide | Expose
87/18/2  \end{zed}

89/19/1  \begin{schema}{Control}
91/19/3  title : String
92/19/5  \\
93/19/3  control : HideExpose
94/19/5  \\
95/19/3  xylimits : Pixel \cross Pixel
96/19/4  \where
97/19/3  (first \ xylimits) \leq (second \
xylimits)
90/19/2  \end{schema}

98/20/1  \begin{zed}
100/20/3 Info
101/20/4 ==
102/21/3 Map
103/21/5 \land
104/21/3 Control
99/20/2  \end{zed}

108/22/1 \begin{zed}
110/22/4 [
111/22/3 Window
112/22/4 ]
109/22/2 \end{zed}

113/23/1 \begin{axdef}
115/23/3 Undefined : Window
114/23/2 \end{axdef}

116/24/1 \begin{schema}{WMZero}
118/24/3 windows : \finset Window
119/24/5 \\
120/24/3 current : Window
121/24/5 \\
122/24/3 contents : Window \pfun Info
123/24/4 \where
124/24/3 Undefined \notin windows
125/24/5 \\
126/24/3 windows = \dom contents
127/24/5 \\
128/24/3 current \in windows \cup \{ Undefined \}
117/24/2 \end{schema}

129/25/1 \begin{schema}{WMOne}
131/25/3 WMZero
132/25/5 \\
133/25/3 maps : Window \pfun Pixmap
134/25/5 \\
135/25/3 areas : Window \pfun (\power Pixel)
136/25/5 \\
137/25/3 screen : Pixmap
138/25/5 \\
139/25/3 background : Pixmap
140/25/4 \where
141/25/3 maps = contents \comp
142/25/3 (\lambda Info @ map)
143/25/5 \\
144/25/3 areas = contents \comp
145/25/3 (\lambda Info @ area)
146/25/5 \\
147/25/3 \disjoint areas
148/25/5 \\
149/25/3 \bigcup (\ran areas) \subseteq \dom
background
150/25/5 \\
151/25/3 screen = background \oplus \bigcup (\ran
maps)
130/25/2 \end{schema}

152/26/1 \begin{axdef}
154/26/3 MaxWindows : \nat
155/26/5 \where
156/26/3 MaxWindows = 20
153/26/2 \end{axdef}

157/27/1 \begin{schema}{WMTwo}
159/27/3 WMOne
160/27/4 \where
161/27/3 \sharp windows \leq MaxWindows
158/27/2 \end{schema}

162/28/1 \begin{axdef}
164/28/3 WINDOWS : Window \pfun Info
163/28/2 \end{axdef}

165/29/1 \begin{schema}{WM}
167/29/3 WMTwo
168/29/5 \\
169/29/3 adjust : WINDOWS \fun WINDOWS
170/29/4 \where
172/29/3 \forall w,w' : WINDOWS | w' = adjust(w)
@ \\ \t2\sharp w' = \sharp w \land \\ \t2w' \comp
173/29/3 (\lambda Info @ \Theta Control) = w'
\comp (\lambda Info @ \Theta Control)
166/29/2 \end{schema}

174/30/1 \begin{schema}{InitWM}
176/30/3 WM'
177/30/4 \where
178/30/3 windows' = \emptyset
179/30/5 \\
```

Fig. A.59: Detailed list of all points of interest for the *BB*-, *Petrol*-, *Elevator*- and *WM*-specification (part 6 of 8). The figure contains the list of points of interest and the annotated listing of the specification.

```
180/30/3  current' = undefined                    234/34/3  xylimits' = (minxy?, maxxy?)      290/39/5  \\
175/30/2  \end{schema}                            221/34/2  \end{schema}                      291/39/3  title' = s?
                                                                                              292/39/5  \\
181/31/1  \begin{schema}{PhiCurrent}              235/35/1  \begin{schema}{GetDimensions}     293/39/3  control' = control
183/31/3  \Delta WM                               237/35/3  PhiCurrent                        294/39/5  \\
184/31/5  \\                                      238/35/5  \\                                295/39/3  xylimits' = xylimits
185/31/3  \Delta Info                             239/35/3  wh! : Pixel                      284/39/2  \end{schema}
186/31/4  \where                                  240/35/5  \\
187/31/3  current \in windows                     241/35/3  xy1 : Pixel                       296/40/1  \begin{schema}{ClearWindow}
188/31/5  \\                                      242/35/5  \\                                298/40/3  \Delta PhiCurrent
189/31/3  current' = current                      243/35/3  xy2 : Pixel                       299/40/4  \where
190/31/5  \\                                      244/35/4  \where                            300/40/3  header' = header
191/31/3  \Theta Info = contents (current)        245/35/3  \Theta Info' = \Theta Info        301/40/5  \\
192/31/5  \\                                      246/35/5  \\                                302/40/3  body' = setval \ White \ body
193/31/3  contents' = adjust ( contents \bigoplus 247/35/3  \dom body = xy1 \dots xy2         303/40/5  \\
\{current \mapsto \Theta Info' \} )               248/35/5  \\                                304/40/3  title' = title
182/31/2  \end{schema}                            249/35/3  wh! = xy2 - xy1                  305/40/5  \\
                                                  236/35/2  \end{schema}                      306/40/3  control' = control
194/32/1  \begin{schema}{NewWindow}                                                           307/40/5  \\
196/32/3  \Delta WM                               250/36/1  \begin{schema}{HideMe}           308/40/3  xylimits' = xylimits
197/32/5  \\                                      252/36/3  PhiCurrent                        297/40/2  \end{schema}
198/32/3  w! : Window                             253/36/4  \where
199/32/5  \\                                      254/36/3  map' = map                        309/41/1  \begin{axdef}
200/32/3  Info                                    255/36/5  \\                                311/41/3  Null : Window
201/32/4  \where                                  256/36/3  title' = title                    310/41/2  \end{axdef}
202/32/3  \sharp windows < MaxWindows             257/36/5  \\
203/32/5  \\                                      258/36/3  control' = Hide                   312/42/1  \begin{schema}{WMErr}
204/32/3  w! \notin windows \cup \{ Undefined \}  259/36/5  \\                                314/42/3  WM
205/32/5  \\                                      260/36/3  xylimits' = xylimits              315/42/4  \where
206/32/3  current' = w!                           251/36/2  \end{schema}                      316/42/3  Null \notin windows
207/32/5  \\                                                                                313/42/2  \end{schema}
208/32/3  control = Expose                        261/37/1  \begin{schema}{ExposeMe}
209/32/5  \\                                      263/37/3  PhiCurrent                        317/43/1  \begin{schema}{SuccessWM}
210/32/3  contents' = adjust ( contents \cup \{ w! 264/37/4  \where                           319/43/3  \Delta WMErr
\mapsto \Theta Info \} )                          265/37/3  map' = map                        320/43/5  \\
195/32/2  \end{schema}                            266/37/5  \\                                321/43/3  w! : Window
                                                  267/37/3  title' = title                    322/43/4  \where
211/33/1  \begin{schema}{DeleteWindow}            268/37/5  \\                                323/43/3  w! \neq Null
213/33/3  \Delta WM                               269/37/3  control' = Expose                 318/43/2  \end{schema}
214/33/4  \where                                  270/37/5  \\
215/33/3  current \in windows                     271/37/3  xylimits' = xylimits              324/44/1  \begin{schema}{TooManyWindows}
216/33/5  \\                                      262/37/2  \end{schema}                      326/44/3  \#i WMErr
217/33/3  current' = Undefined                                                                327/44/5  \\
218/33/5  \\                                      272/38/1  \begin{schema}{SelectWindow}      328/44/3  w! : Window
219/33/3  contents' = adjust ( \{current\} \ndres 274/38/3  \Delta WM                         329/44/4  \where
contents )                                        275/38/5  \\                                330/44/3  \sharp windows \geq MaxWindows
212/33/2  \end{schema}                            276/38/3  w? : Window                     331/44/5  \\
                                                  277/38/4  \where                            332/44/3  w! = Null
220/34/1  \begin{schema}{SetDimensions}           278/38/3  w? \in windows                    325/44/2  \end{schema}
PhiCurrent                                        279/38/5  \\
222/34/3  \\                                      280/38/3  current' = w?
223/34/5  minxy? : Pixel                          281/38/5  \\
224/34/3  \\                                      282/38/3  contents' = contents              333/45/1  \begin{zed}
225/34/5  maxxy? : Pixel                          273/38/2  \end{schema}                      335/45/3  NewWindow1
226/34/3  \where                                                                              336/45/4  ==
227/34/4  map' = map                              283/39/1  \begin{schema}{SetTitle}          337/46/7  (
228/34/3  \\                                      285/39/3  PhiCurrent                        338/46/3  NewWindow
229/34/5  title' = title                          286/39/5  \\                                339/46/5  \land
230/34/3  \\                                      287/39/3  s? : String                       340/46/3  SuccessWM
231/34/5  control' = control                      288/39/4  \where                            343/46/8  )
232/34/3  \\                                      289/39/3  map' = map                        344/47/5  \lor
233/34/5  \\                                                                                345/47/3  TooManyWindows
                                                                                            334/45/2  \end{zed}
```

Fig. A.60: Detailed list of all points of interest for the *BB*-, *Petrol*-, *Elevator*- and *WM*-specification (part 7 of 8). The figure contains the list of points of interest and the annotated listing of the specification.

```
350/48/1  \begin{schema}[NoCurrentWindow]
352/48/3  \Xi WMErr
353/48/4  \where
354/48/3  current = Undefined
351/48/2  \end{schema}

355/49/1  \begin{zed}
357/49/3  DeleteWindow1
358/49/4  ==
359/50/3  DeleteWindow
360/50/5  \lor
361/50/3  NoCurrentWindow
356/49/2  \end{zed}
365/51/1  \begin{zed}
367/51/3  SetDimensions1
368/51/4  ==
369/52/3  SetDimensions
370/52/5  \lor
371/52/3  NoCurrentWindow
366/51/2  \end{zed}
375/53/1  \begin{zed}
377/53/3  GetDimensions1
378/53/4  ==
379/54/3  GetDimensions
380/54/5  \lor
381/54/3  NoCurrentWindow
376/53/2  \end{zed}
385/55/1  \begin{zed}
387/55/3  SetTitle1
388/55/4  ==
389/56/3  SetTitle
390/56/5  \lor
391/56/3  NoCurrentWindow
386/55/2  \end{zed}
395/57/1  \begin{zed}
397/57/3  ClearWindow1
398/57/4  ==
399/58/3  ClearWindow
400/58/5  \lor
401/58/3  NoCurrentWindow
396/57/2  \end{zed}

405/59/1  \begin{schema}[InvalidWindow]
407/59/3  \Xi WMErr
408/59/5  \\
409/59/3  w? : Window
410/59/4  \where
411/59/3  w? \notin windows
406/59/2  \end{schema}

412/60/1  \begin{zed}
414/60/3  SelectWindow1
415/60/4  ==
416/61/3  SelectWindow
417/61/5  \lor
418/61/3  InvalidWindow
413/60/2  \end{zed}

422/62/1  \begin{schema}[ITC]
424/62/3  hosts : \power String

425/62/5  \\
426/62/3  wms : String \pfun WM
427/62/4  \where
428/62/3  \dom wms \subseteq hosts
429/62/5  \\
430/62/3  \disjoint ( wms \comp
431/62/3  ( \lambda WM @ windows ) )
423/62/2  \end{schema}

432/63/1  \begin{schema}[InitITC]
434/63/3  ITC'
435/63/4  \where
436/63/3  hosts' = \emptyset
433/63/2  \end{schema}

437/64/1  \begin{zed}
439/64/4  [
440/64/3  EMPTYSTRING
441/64/4  ]
438/64/2  \end{zed}

442/65/1  \begin{schema}[AddHost]
444/65/3  \Delta ITC
445/65/5  \\
446/65/3  host? : String
447/65/4  \where
448/65/3  host? \notin hosts \cup
EMPTYSTRING
449/65/5  \\
450/65/3  hosts' = hosts \cup \{ host? \}
451/65/5  \\
452/65/3  wms' = wms
443/65/2  \end{schema}

453/66/1  \begin{schema}[RemoveHost]
455/66/3  \Delta ITC
456/66/5  \\
457/66/3  host? : String
458/66/4  \where
459/66/3  host? in hosts
460/66/5  \\
461/66/3  hosts' = hosts \setminus \{ host? \}
462/66/5  \\
463/66/3  wms' = wms
454/66/2  \end{schema}

464/67/1  \begin{schema}[PhiHost]
466/67/3  \Delta ITC
467/67/5  \\
468/67/3  localhost : String
469/67/4  \where
470/67/3  hosts' = hosts
471/67/5  \\
472/67/3  localhosts \in hosts
465/67/2  \end{schema}

473/68/1  \begin{schema}[ExecWM]
475/68/3  PhiHost
476/68/5  \\
477/68/3  initwm : WM

478/68/4  \where
479/68/3  localhost \notin \dom wms
480/68/5  \\
481/68/3  wms' = wms \cup \{ localhost \mapsto
initwm \}
474/68/2  \end{schema}

482/69/1  \begin{schema}[KillWM]
484/69/3  PhiHost
485/69/4  \where
486/69/3  localhost \in \dom wms
487/69/5  \\
488/69/3  wms' = \{ localhost \} \ndres wms
483/69/2  \end{schema}

489/70/1  \begin{schema}[PhiWM]
491/70/3  PhiHost
492/70/5  \\
493/70/3  \Delta WM
494/70/5  \\
495/70/3  host : String
496/70/4  \where
497/70/3  \Theta WM = wms \ host
498/70/5  \\
499/70/3  \Theta WM' = wms' \ host
490/70/2  \end{schema}

500/71/1  \begin{schema}[NewWindowITC]
502/71/3  NewWindow1
503/71/5  \\
504/71/3  PhiWM
505/71/5  \\
506/71/3  host? : String
507/71/4  \where
508/71/3  host? = EMPTYSTRING \imp host =
localhost
509/71/5  \\
510/71/3  host? \neq EMPTYSTRING \imp host = host?
501/71/2  \end{schema}

511/72/1  \begin{zed}
513/72/3  DeleteWindowITC
514/72/4  ==
515/73/3  DeleteWindow1
516/73/5  \land
517/73/3  PhiWM
512/72/2  \end{zed}
```

Fig. A.61: Detailed list of all points of interest for the *BB*-, *Petrol*-, *Elevator*- and *WM*-specification (part 8 of 8). The figure contains the list of points of interest and the annotated listing of the specification.

Spec.	Deps.	V-ID	V	A	S/E	P	CC	v'(i)	v'(u)	DU
BB Total			**72**	**267**	**11**	**27**	**24**	**4**	**10**	**4**
BB	SCD	18	54	190	8	20	17	4	7	4
BB	SCD	23	26	43	3	4	4	1	1	0
BB	SCD	32	38	97	6	9	7	1	1	0
BB	SCD	34	54	190	8	20	17	4	7	4
BB	SCD	41	38	95	6	9	7	1	1	0
BB	SCD	43	54	190	8	20	17	4	7	4
BB	SCD	48	66	232	10	25	22	4	10	4
Mean			**47,143**	**148,143**	**7,000**	**15,286**	**13,000**	**2,714**	**4,857**	**2,286**
BB	SD	18	50	168	8	18	15	2	3	4
BB	SD	23	24	43	3	4	4	1	1	0
BB	SD	32	37	97	6	9	7	1	1	0
BB	SD	34	49	155	8	16	13	1	1	2
BB	SD	41	38	95	6	9	7	1	1	0
BB	SD	43	50	157	8	16	13	1	1	2
BB	SD	48	37	87	6	9	6	1	1	0
Mean			**40,714**	**114,571**	**6,429**	**11,571**	**9,286**	**1,143**	**1,286**	**1,143**
BB	SC	18	14	44	2	5	5	1	1	0
BB	SC	23	16	43	3	4	4	1	1	0
BB	SC	32	31	91	6	9	7	1	1	0
BB	SC	34	52	175	8	19	16	4	4	1
BB	SC	41	37	93	6	9	7	1	1	0
BB	SC	43	52	171	8	18	15	4	4	1
BB	SC	48	63	202	10	22	19	4	4	0
Mean			**37,857**	**117,000**	**6,143**	**12,286**	**10,429**	**2,286**	**2,286**	**0,286**
Petrol Total			**134**	**674**	**15**	**57**	**53**	**10**	**28**	**131**
Petrol	SCD	17	105	536	11	46	44	10	28	110
Petrol	SCD	19	105	536	11	46	44	10	28	110
Petrol	SCD	28	114	575	13	50	47	10	28	115
Petrol	SCD	30	115	577	13	50	47	10	28	115
Petrol	SCD	32	116	579	13	50	47	10	28	115
Petrol	SCD	41	107	536	11	46	44	10	28	110
Petrol	SCD	43	107	536	11	46	44	10	28	110
Petrol	SCD	45	107	536	11	46	44	10	28	110
Petrol	SCD	49	107	536	11	46	44	10	28	110
Petrol	SCD	51	107	536	11	46	44	10	28	110
Petrol	SCD	53	107	536	11	46	44	10	28	110
Petrol	SCD	56	33	25	1	1	1	1	1	0
Petrol	SCD	62	107	536	11	46	44	10	28	110
Petrol	SCD	64	107	536	11	46	44	10	28	110
Petrol	SCD	66	107	536	11	46	44	10	28	110
Petrol	SCD	70	107	536	11	46	44	10	28	110
Petrol	SCD	72	107	536	11	46	44	10	28	110
Petrol	SCD	74	107	536	11	46	44	10	28	110
Petrol	SCD	77	32	5	1	1	1	1	1	0
Petrol	SCD	125	43	38	3	4	2	1	1	0
Petrol	SCD	127	48	51	4	5	3	1	1	0
Mean			**95,000**	**445,429**	**9,619**	**38,333**	**36,381**	**8,286**	**22,857**	**89,762**
Petrol	SD	17	89	410	11	38	36	2	4	73
Petrol	SD	19	90	412	11	38	36	2	4	73
Petrol	SD	28	98	437	13	41	38	1	1	73
Petrol	SD	30	99	439	13	41	38	1	1	73
Petrol	SD	32	100	441	13	41	38	1	1	73
Petrol	SD	41	92	413	11	38	36	2	4	73
Petrol	SD	43	93	415	11	38	36	2	4	73
Petrol	SD	45	95	419	11	38	36	2	4	73
Petrol	SD	49	94	404	11	37	35	1	1	68
Petrol	SD	51	94	404	11	37	35	1	1	68
Petrol	SD	53	94	404	11	37	35	1	1	68
Petrol	SD	56	30	19	1	1	1	1	1	0
Petrol	SD	62	96	420	11	38	36	2	4	73
Petrol	SD	64	97	424	11	38	36	2	4	75

Fig. A.62: Detailed list of response variables for all points of interest (part 1 of 11). The figure provides the specification name, the abstraction criterion, the unique vertex ID and the full list of ASRN and specification measures.

Spec.	Deps.	V-ID	V	A	S/E	P	CC	v'(I)	v'(u)	DU
Petrol	SD	66	99	426	11	38	36	2	4	73
Petrol	SD	70	98	412	11	37	35	1	1	68
Petrol	SD	72	98	412	11	37	35	1	1	68
Petrol	SD	74	98	412	11	37	35	1	1	68
Petrol	SD	77	32	5	1	1	1	1	1	0
Petrol	SD	125	43	38	3	4	2	1	1	0
Petrol	SD	127	48	51	4	5	3	1	1	0
Mean			**84,619**	**343,667**	**9,619**	**31,429**	**29,476**	**1,381**	**2,143**	**57,762**
Petrol	SC	17	14	44	2	5	5	1	1	0
Petrol	SC	19	16	48	2	5	5	1	1	0
Petrol	SC	28	26	75	5	7	6	1	1	0
Petrol	SC	30	28	83	5	8	7	1	1	0
Petrol	SC	32	29	85	5	8	7	1	1	0
Petrol	SC	41	31	93	4	10	10	1	1	0
Petrol	SC	43	33	101	4	11	11	1	1	0
Petrol	SC	45	31	80	4	7	7	1	1	0
Petrol	SC	49	39	134	4	14	14	4	4	2
Petrol	SC	51	40	133	4	14	14	4	4	1
Petrol	SC	53	41	136	4	14	14	4	4	2
Petrol	SC	56	18	9	1	1	1	1	1	0
Petrol	SC	62	34	75	3	8	8	1	1	0
Petrol	SC	64	32	59	3	5	5	1	1	0
Petrol	SC	66	34	63	3	5	5	1	1	0
Petrol	SC	70	43	116	3	13	13	5	5	2
Petrol	SC	72	44	119	3	13	13	5	5	1
Petrol	SC	74	45	121	3	13	13	5	5	3
Petrol	SC	77	25	5	1	1	1	1	1	0
Petrol	SC	125	36	38	3	4	2	1	1	0
Petrol	SC	127	41	51	4	5	3	1	1	0
Mean			**32,381**	**79,429**	**3,333**	**8,143**	**7,810**	**2,000**	**2,000**	**0,524**
WM Total			**520**	**2633**	**73**	**233**	**213**	**39**	**544**	**215**
WM	SCD	30	16	42	4	7	7	1	1	0
WM	SCD	32	17	44	4	7	7	1	1	0
WM	SCD	83	21	63	4	8	7	1	1	0
WM	SCD	85	25	66	5	8	7	1	1	0
WM	SCD	97	22	49	4	6	5	1	1	0
WM	SCD	124	19	35	3	4	4	1	1	0
WM	SCD	126	284	1271	48	112	103	28	232	103
WM	SCD	128	284	1271	48	112	103	28	232	103
WM	SCD	141	284	1271	48	112	103	28	232	103
WM	SCD	142	49	25	1	1	1	1	1	0
WM	SCD	144	284	1271	48	112	103	28	232	103
WM	SCD	145	49	25	1	1	1	1	1	0
WM	SCD	147	58	50	3	4	4	1	1	0
WM	SCD	149	62	61	4	6	6	1	1	0
WM	SCD	151	60	62	3	6	6	1	1	0
WM	SCD	156	50	11	1	2	2	1	1	0
WM	SCD	161	68	72	5	7	7	2	2	1
WM	SCD	172	50	9	1	1	1	1	1	0
WM	SCD	173	50	9	1	1	1	1	1	0
WM	SCD	176	50	11	1	1	1	1	1	0
WM	SCD	178	60	48	3	4	4	2	2	0
WM	SCD	180	60	48	3	4	4	2	2	0
WM	SCD	185	285	1273	48	112	103	28	232	103
WM	SCD	187	285	1273	48	112	103	28	232	103
WM	SCD	189	285	1273	48	112	103	28	232	103
WM	SCD	191	285	1273	48	112	103	28	232	103
WM	SCD	193	285	1273	48	112	103	28	232	103
WM	SCD	200	285	1273	48	112	103	28	232	103
WM	SCD	202	79	122	8	11	9	2	2	1
WM	SCD	204	288	1304	48	113	104	28	250	104
WM	SCD	206	287	1277	48	112	103	28	232	103
WM	SCD	208	287	1277	48	112	103	28	232	103

Fig. A.63: Detailed list of response variables for all points of interest (part 2 of 11). The figure provides the specification name, the abstraction criterion, the unique vertex ID and the full list of ASRN and specification measures.

Spec.	Deps.	V-ID	V	A	S/E	P	CC	v'(i)	v'(u)	DU
WM	SCD	210	287	1277	48	112	103	28	232	103
WM	SCD	215	287	1277	48	112	103	28	232	103
WM	SCD	217	287	1277	48	112	103	28	232	103
WM	SCD	219	287	1277	48	112	103	28	232	103
WM	SCD	228	292	1350	48	117	108	28	268	115
WM	SCD	230	310	1418	49	120	111	28	286	119
WM	SCD	232	298	1299	48	112	103	28	232	103
WM	SCD	234	304	1343	48	116	107	28	250	103
WM	SCD	245	300	1303	48	112	103	28	232	103
WM	SCD	247	300	1303	48	112	103	28	232	103
WM	SCD	249	304	1338	48	114	105	28	251	103
WM	SCD	254	307	1380	48	117	108	28	268	115
WM	SCD	256	314	1426	49	120	111	28	286	119
WM	SCD	258	82	63	5	6	5	1	1	0
WM	SCD	260	314	1427	48	120	111	28	286	119
WM	SCD	265	311	1388	48	117	108	28	268	115
WM	SCD	267	318	1434	49	120	111	28	286	119
WM	SCD	269	86	65	5	6	5	1	1	0
WM	SCD	271	314	1427	48	120	111	28	286	119
WM	SCD	278	95	105	6	10	9	2	2	1
WM	SCD	280	306	1315	48	112	103	28	232	103
WM	SCD	282	306	1315	48	112	103	28	232	103
WM	SCD	289	311	1388	48	117	108	28	268	115
WM	SCD	291	314	1362	49	116	107	28	250	103
WM	SCD	293	306	1315	48	112	103	28	232	103
WM	SCD	295	314	1427	48	120	111	28	286	119
WM	SCD	300	309	1345	48	114	105	28	250	103
WM	SCD	302	307	1317	48	112	103	28	232	103
WM	SCD	304	319	1436	49	120	111	28	286	119
WM	SCD	306	307	1317	48	112	103	28	232	103
WM	SCD	308	315	1429	48	120	111	28	286	119
WM	SCD	316	90	71	5	7	7	2	2	1
WM	SCD	323	314	1360	49	115	106	28	250	103
WM	SCD	330	99	101	7	10	9	2	2	1
WM	SCD	332	312	1352	48	114	105	28	250	103
WM	SCD	354	310	1323	48	112	103	28	232	103
WM	SCD	411	99	101	6	10	9	2	2	1
WM	SCD	428	400	1746	60	150	140	39	277	170
WM	SCD	430	400	1746	60	150	140	39	277	170
WM	SCD	431	90	13	1	1	1	1	1	0
WM	SCD	434	90	9	1	1	1	1	1	0
WM	SCD	436	100	42	3	4	4	2	2	0
WM	SCD	448	400	1746	60	150	140	39	277	170
WM	SCD	450	400	1746	60	150	140	39	277	170
WM	SCD	452	400	1746	60	150	140	39	277	170
WM	SCD	459	400	1746	60	150	140	39	277	170
WM	SCD	461	400	1746	60	150	140	39	277	170
WM	SCD	463	400	1746	60	150	140	39	277	170
WM	SCD	470	400	1746	60	150	140	39	277	170
WM	SCD	472	400	1746	60	150	140	39	277	170
WM	SCD	479	400	1746	60	150	140	39	277	170
WM	SCD	481	400	1746	60	150	140	39	277	170
WM	SCD	486	400	1746	60	150	140	39	277	170
WM	SCD	488	400	1746	60	150	140	39	277	170
WM	SCD	497	400	1746	60	150	140	39	277	170
WM	SCD	499	400	1746	60	150	140	39	277	170
WM	SCD	502	94	20	2	2	2	1	1	0
WM	SCD	508	120	109	7	10	9	1	1	0
WM	SCD	510	117	96	6	9	8	1	1	0
WM	SCD	515	101	36	3	4	3	1	1	0
Mean			**236,087**	**958,087**	**34,717**	**82,598**	**76,413**	**20,620**	**167,217**	**80,815**
WM	SD	30	16	42	4	7	7	1	1	0
WM	SD	32	17	44	4	7	7	1	1	0

Fig. A.64: Detailed list of response variables for all points of interest (part 3 of 11). The figure provides the specification name, the abstraction criterion, the unique vertex ID and the full list of ASRN and specification measures.

Spec.	Deps.	V-ID	V	A	S/E	P	CC	v'(I)	v'(u)	DU
WM	SD	83	21	63	4	8	7	1	1	0
WM	SD	85	25	66	5	8	7	1	1	0
WM	SD	97	22	49	4	6	5	1	1	0
WM	SD	124	19	35	3	4	4	1	1	0
WM	SD	126	110	382	21	38	33	2	10	36
WM	SD	128	83	246	15	26	22	2	5	8
WM	SD	141	119	403	23	40	35	2	10	36
WM	SD	142	21	11	1	1	1	1	1	0
WM	SD	144	121	407	23	40	35	2	10	36
WM	SD	145	23	15	1	1	1	1	1	0
WM	SD	147	33	42	3	4	4	1	1	0
WM	SD	149	38	55	4	6	6	1	1	0
WM	SD	151	39	62	3	6	6	1	1	0
WM	SD	156	29	11	1	2	2	1	1	0
WM	SD	161	47	67	5	6	6	1	1	1
WM	SD	172	30	9	1	1	1	1	1	0
WM	SD	173	30	9	1	1	1	1	1	0
WM	SD	176	30	11	1	1	1	1	1	0
WM	SD	178	39	43	3	3	3	1	1	0
WM	SD	180	39	43	3	3	3	1	1	0
WM	SD	185	121	374	21	37	32	2	4	34
WM	SD	187	96	256	15	27	23	2	2	10
WM	SD	189	93	232	15	24	20	1	1	4
WM	SD	191	124	391	21	37	32	2	4	42
WM	SD	193	123	370	21	36	31	1	1	32
WM	SD	200	125	381	21	37	32	2	3	34
WM	SD	202	63	111	8	10	8	1	1	1
WM	SD	204	66	120	8	11	9	1	1	1
WM	SD	206	59	100	6	9	7	1	1	0
WM	SD	208	110	259	19	24	18	1	1	17
WM	SD	210	129	380	21	36	31	1	1	32
WM	SD	215	105	272	15	27	23	2	2	10
WM	SD	217	62	84	6	8	7	1	1	0
WM	SD	219	130	382	21	36	31	1	1	32
WM	SD	228	84	158	12	15	12	1	1	12
WM	SD	230	109	227	15	20	16	1	1	16
WM	SD	232	106	211	15	19	15	1	1	12
WM	SD	234	78	105	7	11	9	1	1	0
WM	SD	245	142	382	21	36	31	1	1	32
WM	SD	247	102	164	13	18	15	1	1	1
WM	SD	249	72	67	4	8	7	1	1	0
WM	SD	254	101	178	12	15	12	1	1	12
WM	SD	256	116	233	15	20	16	1	1	16
WM	SD	258	73	63	5	6	5	1	1	0
WM	SD	260	119	239	15	21	17	1	1	16
WM	SD	265	105	184	12	15	12	1	1	12
WM	SD	267	120	241	15	20	16	1	1	16
WM	SD	269	77	65	5	6	5	1	1	0
WM	SD	271	119	239	15	21	17	1	1	16
WM	SD	278	86	98	6	9	8	1	1	1
WM	SD	280	82	85	5	8	7	1	1	0
WM	SD	282	152	392	21	36	31	1	1	32
WM	SD	289	106	184	12	15	12	1	1	12
WM	SD	291	89	105	7	10	8	1	1	0
WM	SD	293	118	225	15	19	15	1	1	12
WM	SD	295	120	239	15	21	17	1	1	16
WM	SD	300	87	97	7	9	7	1	1	0
WM	SD	302	93	109	9	11	9	1	1	0
WM	SD	304	122	243	15	20	16	1	1	16
WM	SD	306	119	227	15	19	15	1	1	12
WM	SD	308	121	241	15	21	17	1	1	16
WM	SD	316	81	66	5	6	6	1	1	1
WM	SD	323	85	77	6	8	6	1	1	0
WM	SD	330	91	96	7	9	8	1	1	1

Fig. A.65: Detailed list of response variables for all points of interest (part 4 of 11). The figure provides the specification name, the abstraction criterion, the unique vertex ID and the full list of ASRN and specification measures.

Spec.	Deps.	V-ID	V	A	S/E	P	CC	v'(i)	v'(u)	DU
WM	SD	332	85	68	5	7	6	1	1	0
WM	SD	354	172	402	24	39	31	2	3	9
WM	SD	411	98	96	6	9	8	1	1	1
WM	SD	428	138	260	11	27	26	2	3	35
WM	SD	430	128	198	10	19	18	2	2	23
WM	SD	431	83	13	1	1	1	1	1	0
WM	SD	434	83	9	1	1	1	1	1	0
WM	SD	436	92	37	3	3	3	1	1	0
WM	SD	448	116	144	7	13	13	2	2	13
WM	SD	450	110	118	6	11	11	1	1	9
WM	SD	452	128	189	10	18	17	1	1	18
WM	SD	459	112	132	6	12	12	2	2	13
WM	SD	461	111	120	6	11	11	1	1	9
WM	SD	463	129	191	10	18	17	1	1	18
WM	SD	470	111	120	6	11	11	1	1	9
WM	SD	472	113	133	6	12	12	2	2	13
WM	SD	479	132	208	10	19	18	2	2	23
WM	SD	481	131	195	10	18	17	1	1	18
WM	SD	486	133	210	10	19	18	2	2	23
WM	SD	488	132	197	10	18	17	1	1	18
WM	SD	497	134	212	10	19	18	2	2	23
WM	SD	499	108	78	5	8	7	1	1	1
WM	SD	502	93	20	2	2	2	1	1	0
WM	SD	508	119	107	7	10	9	1	1	0
WM	SD	510	116	94	6	9	8	1	1	0
WM	SD	515	101	36	3	4	3	1	1	0
			91,739	**159,283**	**9,598**	**15,087**	**13,076**	**1,196**	**1,565**	**10,022**
WM	SC	30	16	42	4	7	7	1	1	0
WM	SC	32	17	44	4	7	7	1	1	0
WM	SC	83	21	63	4	8	7	1	1	0
WM	SC	85	25	66	5	8	7	1	1	0
WM	SC	97	22	49	4	6	5	1	1	0
WM	SC	124	19	35	3	4	4	1	1	0
WM	SC	126	22	47	3	5	5	1	1	0
WM	SC	128	24	49	3	5	5	1	1	0
WM	SC	141	33	72	5	7	7	1	1	0
WM	SC	142	15	11	1	1	1	1	1	0
WM	SC	144	35	76	5	7	7	1	1	0
WM	SC	145	17	15	1	1	1	1	1	0
WM	SC	147	27	42	3	4	4	1	1	0
WM	SC	149	32	55	4	6	6	1	1	0
WM	SC	151	33	62	3	6	6	1	1	0
WM	SC	156	23	11	1	2	2	1	1	0
WM	SC	161	36	54	4	5	5	1	1	0
WM	SC	172	23	9	1	1	1	1	1	0
WM	SC	173	23	9	1	1	1	1	1	0
WM	SC	176	23	9	1	1	1	1	1	0
WM	SC	178	33	46	3	4	4	2	2	0
WM	SC	180	34	48	3	4	4	2	2	0
WM	SC	185	28	20	2	2	2	1	1	0
WM	SC	187	36	54	3	5	5	1	1	0
WM	SC	189	156	522	27	60	54	20	20	4
WM	SC	191	44	70	4	6	6	1	1	0
WM	SC	193	161	542	28	62	56	20	20	7
WM	SC	200	47	60	5	6	4	1	1	0
WM	SC	202	56	94	7	9	7	1	1	0
WM	SC	204	133	402	20	48	45	19	19	0
WM	SC	206	135	406	20	48	45	19	19	1
WM	SC	208	65	109	8	10	7	1	1	0
WM	SC	210	140	424	21	50	47	19	19	3
WM	SC	215	60	81	5	8	7	1	1	0
WM	SC	217	153	443	24	52	47	17	17	3
WM	SC	219	158	458	25	54	49	17	17	3

Fig. A.66: Detailed list of response variables for all points of interest (part 5 of 11). The figure provides the specification name, the abstraction criterion, the unique vertex ID and the full list of ASRN and specification measures.

Spec.	Deps.	V-ID	V	A	S/E	P	CC	v'(I)	v'(u)	DU
WM	SC	228	168	503	27	57	51	19	19	0
WM	SC	230	175	514	28	58	52	19	19	0
WM	SC	232	174	508	28	58	52	19	19	0
WM	SC	234	175	519	27	59	53	19	19	0
WM	SC	245	177	528	27	60	54	20	20	3
WM	SC	247	78	112	8	12	10	1	1	0
WM	SC	249	180	536	27	61	55	20	20	0
WM	SC	254	67	62	5	6	5	1	1	0
WM	SC	256	70	68	5	6	5	1	1	0
WM	SC	258	69	63	5	6	5	1	1	0
WM	SC	260	70	64	5	6	5	1	1	0
WM	SC	265	70	62	5	6	5	1	1	0
WM	SC	267	73	68	5	6	5	1	1	0
WM	SC	269	72	63	5	6	5	1	1	0
WM	SC	271	73	64	5	6	5	1	1	0
WM	SC	278	78	81	5	8	7	1	1	0
WM	SC	280	143	345	16	42	41	17	17	1
WM	SC	282	144	348	16	42	41	17	17	3
WM	SC	289	188	505	27	57	51	19	19	0
WM	SC	291	195	522	28	59	53	19	19	0
WM	SC	293	193	510	28	58	52	19	19	0
WM	SC	295	191	507	27	57	51	19	19	0
WM	SC	300	192	507	27	57	51	19	19	0
WM	SC	302	199	521	29	59	53	19	19	0
WM	SC	304	199	520	28	58	52	19	19	0
WM	SC	306	198	514	28	58	52	19	19	0
WM	SC	308	196	511	27	57	51	19	19	0
WM	SC	316	85	53	4	5	5	1	1	0
WM	SC	323	173	422	21	49	46	19	19	0
WM	SC	330	94	83	6	8	7	1	1	0
WM	SC	332	171	414	20	48	45	19	19	0
WM	SC	354	128	198	14	20	15	1	1	0
WM	SC	411	94	83	5	8	7	1	1	0
WM	SC	428	83	36	2	4	4	1	1	0
WM	SC	430	83	32	2	3	3	1	1	0
WM	SC	431	77	13	1	1	1	1	1	0
WM	SC	434	77	9	1	1	1	1	1	0
WM	SC	436	87	42	3	4	4	2	2	0
WM	SC	448	94	65	4	6	6	1	1	0
WM	SC	450	96	76	4	7	7	2	2	1
WM	SC	452	98	81	4	8	8	2	2	0
WM	SC	459	92	51	3	5	5	1	1	0
WM	SC	461	94	62	3	6	6	2	2	1
WM	SC	463	96	67	3	7	7	2	2	0
WM	SC	470	174	369	17	45	44	20	20	2
WM	SC	472	95	45	3	4	4	1	1	0
WM	SC	479	100	58	4	5	5	1	1	0
WM	SC	481	104	76	4	7	7	2	2	1
WM	SC	486	102	58	4	5	5	1	1	0
WM	SC	488	104	69	4	6	6	2	2	1
WM	SC	497	108	77	5	8	7	1	1	0
WM	SC	499	179	376	17	45	44	20	20	4
WM	SC	502	94	20	2	2	2	1	1	0
WM	SC	508	120	109	7	10	9	1	1	0
WM	SC	510	117	96	6	9	8	1	1	0
WM	SC	515	101	36	3	4	3	1	1	0
			96,217	**183,696**	**10,207**	**20,489**	**18,707**	**6,554**	**6,554**	**0,413**
Elevator Total			**349**	**3668**	**32**	**159**	**144**	**32**	**1069**	**1212**
Elevator	SCD	21	289	2966	28	130	117	32	895	969
Elevator	SCD	23	289	2966	28	130	117	32	895	969
Elevator	SCD	25	289	2966	28	130	117	32	895	969
Elevator	SCD	27	289	2966	28	130	117	32	895	969
Elevator	SCD	32	294	3002	29	132	119	32	895	989

Fig. A.67: Detailed list of response variables for all points of interest (part 6 of 11). The figure provides the specification name, the abstraction criterion, the unique vertex ID and the full list of ASRN and specification measures.

Spec.	Deps.	V-ID	V	A	S/E	P	CC	v'(i)	v'(u)	DU
Elevator	SCD	34	295	2994	29	132	119	32	895	979
Elevator	SCD	36	296	2997	29	132	119	32	895	980
Elevator	SCD	38	297	2999	29	132	119	32	895	980
Elevator	SCD	40	298	3001	29	132	119	32	895	979
Elevator	SCD	42	322	3341	29	144	131	32	1069	1051
Elevator	SCD	47	314	3055	30	134	120	32	895	999
Elevator	SCD	54	119	63	4	6	5	1	1	0
Elevator	SCD	56	305	2988	28	130	117	32	895	969
Elevator	SCD	58	305	2988	28	130	117	32	895	969
Elevator	SCD	60	305	2988	28	130	117	32	895	969
Elevator	SCD	62	305	2988	28	130	117	32	895	969
Elevator	SCD	64	305	2988	28	130	117	32	895	969
Elevator	SCD	66	317	3304	28	142	129	32	1069	1041
Elevator	SCD	75	119	65	4	6	5	1	1	0
Elevator	SCD	77	305	2988	28	130	117	32	895	969
Elevator	SCD	79	305	2988	28	130	117	32	895	969
Elevator	SCD	81	305	2988	28	130	117	32	895	969
Elevator	SCD	83	305	2988	28	130	117	32	895	969
Elevator	SCD	85	305	2988	28	130	117	32	895	969
Elevator	SCD	87	317	3304	28	142	129	32	1069	1041
Elevator	SCD	102	305	2988	28	130	117	32	895	969
Elevator	SCD	104	305	2988	28	130	117	32	895	969
Elevator	SCD	106	305	2988	28	130	117	32	895	969
Elevator	SCD	108	305	2988	28	130	117	32	895	969
Elevator	SCD	111	305	2988	28	130	117	32	895	969
Elevator	SCD	113	305	2988	28	130	117	32	895	969
Elevator	SCD	115	305	2988	28	130	117	32	895	969
Elevator	SCD	117	305	2988	28	130	117	32	895	969
Elevator	SCD	119	317	3304	28	142	129	32	1069	1041
Elevator	SCD	124	305	2988	28	130	117	32	895	969
Elevator	SCD	126	305	2988	28	130	117	32	895	969
Elevator	SCD	128	305	2988	28	130	117	32	895	969
Elevator	SCD	130	305	2988	28	130	117	32	895	969
Elevator	SCD	133	305	2988	28	130	117	32	895	969
Elevator	SCD	135	305	2988	28	130	117	32	895	969
Elevator	SCD	137	305	2988	28	130	117	32	895	969
Elevator	SCD	139	305	2988	28	130	117	32	895	969
Elevator	SCD	141	317	3304	28	142	129	32	1069	1041
Elevator	SCD	146	305	2988	28	130	117	32	895	969
Elevator	SCD	148	305	2988	28	130	117	32	895	969
Elevator	SCD	150	305	2988	28	130	117	32	895	969
Elevator	SCD	152	305	2988	28	130	117	32	895	969
Elevator	SCD	154	305	2988	28	130	117	32	895	969
Elevator	SCD	156	305	2988	28	130	117	32	895	969
Elevator	SCD	158	305	2988	28	130	117	32	895	969
Elevator	SCD	160	305	2988	28	130	117	32	895	969
Elevator	SCD	162	317	3304	28	142	129	32	1069	1041
Elevator	SCD	167	305	2988	28	130	117	32	895	969
Elevator	SCD	169	305	2988	28	130	117	32	895	969
Elevator	SCD	171	305	2988	28	130	117	32	895	969
Elevator	SCD	173	305	2988	28	130	117	32	895	969
Elevator	SCD	175	305	2988	28	130	117	32	895	969
Elevator	SCD	177	305	2988	28	130	117	32	895	969
Elevator	SCD	179	305	2988	28	130	117	32	895	969
Elevator	SCD	181	305	2988	28	130	117	32	895	969
Elevator	SCD	183	317	3304	28	142	129	32	1069	1041
Elevator	SCD	188	305	2988	28	130	117	32	895	969
Elevator	SCD	190	305	2988	28	130	117	32	895	969
Elevator	SCD	192	305	2988	28	130	117	32	895	969
Elevator	SCD	194	305	2988	28	130	117	32	895	969
Elevator	SCD	196	305	2988	28	130	117	32	895	969
Elevator	SCD	198	305	2988	28	130	117	32	895	969
Elevator	SCD	200	305	2988	28	130	117	32	895	969
Elevator	SCD	202	305	2988	28	130	117	32	895	969

Fig. A.68: Detailed list of response variables for all points of interest (part 7 of 11). The figure provides the specification name, the abstraction criterion, the unique vertex ID and the full list of ASRN and specification measures.

Spec.	Deps.	V-ID	V	A	S/E	P	CC	v'(l)	v'(u)	DU
Elevator	SCD	204	305	2988	28	130	117	32	895	969
Elevator	SCD	206	305	2988	28	130	117	32	895	969
Elevator	SCD	208	317	3304	28	142	129	32	1069	1041
Elevator	SCD	213	305	2988	28	130	117	32	895	969
Elevator	SCD	215	305	2988	28	130	117	32	895	969
Elevator	SCD	217	305	2988	28	130	117	32	895	969
Elevator	SCD	219	305	2988	28	130	117	32	895	969
Elevator	SCD	221	305	2988	28	130	117	32	895	969
Elevator	SCD	223	305	2988	28	130	117	32	895	969
Elevator	SCD	225	305	2988	28	130	117	32	895	969
Elevator	SCD	227	305	2988	28	130	117	32	895	969
Elevator	SCD	229	305	2988	28	130	117	32	895	969
Elevator	SCD	231	317	3304	28	142	129	32	1069	1041
Elevator	SCD	266	305	2988	28	130	117	32	895	969
Elevator	SCD	268	305	2988	28	130	117	32	895	969
Elevator	SCD	270	305	2988	28	130	117	32	895	969
Elevator	SCD	272	305	2988	28	130	117	32	895	969
Elevator	SCD	274	305	2988	28	130	117	32	895	969
Elevator	SCD	276	308	3012	28	133	120	32	904	969
Elevator	SCD	281	305	2988	28	130	117	32	895	969
Elevator	SCD	283	305	2988	28	130	117	32	895	969
Elevator	SCD	285	305	2988	28	130	117	32	895	969
Elevator	SCD	287	305	2988	28	130	117	32	895	969
Elevator	SCD	289	305	2988	28	130	117	32	895	969
Elevator	SCD	291	305	2988	28	130	117	32	895	969
Elevator	SCD	293	308	3012	28	133	120	32	904	969
Elevator	SCD	300	131	104	7	10	5	1	1	0
Elevator	SCD	304	128	93	6	9	5	1	1	0
Elevator	SCD	310	128	95	6	9	5	1	1	0
Elevator	SCD	325	122	73	4	7	5	1	1	0
Elevator	SCD	335	116	50	2	5	4	1	1	0
Elevator	SCD	342	120	58	3	6	5	1	1	0
Elevator	SCD	344	117	50	2	5	4	1	1	0
Mean			**289,059**	**2759,765**	**25,980**	**120,422**	**108,363**	**29,265**	**831,647**	**890,853**
Elevator	SD	21	197	1237	26	87	76	2	42	468
Elevator	SD	23	162	615	26	51	40	2	11	100
Elevator	SD	25	176	836	26	64	53	2	21	229
Elevator	SD	27	177	838	26	64	53	2	21	229
Elevator	SD	32	203	1209	27	87	76	1	1	468
Elevator	SD	34	166	616	26	51	40	1	1	100
Elevator	SD	36	182	829	27	64	53	1	1	229
Elevator	SD	38	183	831	27	64	53	1	1	229
Elevator	SD	40	181	590	27	50	39	1	1	46
Elevator	SD	42	193	646	27	50	39	1	1	82
Elevator	SD	47	223	1072	28	78	66	1	1	338
Elevator	SD	54	89	63	4	6	5	1	1	0
Elevator	SD	56	226	1219	26	85	74	1	1	448
Elevator	SD	58	188	634	25	49	38	1	1	90
Elevator	SD	60	203	844	26	62	51	1	1	218
Elevator	SD	62	203	844	26	62	51	1	1	218
Elevator	SD	64	189	579	26	48	37	1	1	36
Elevator	SD	66	189	611	26	48	37	1	1	72
Elevator	SD	75	90	65	4	6	5	1	1	0
Elevator	SD	77	227	1221	26	85	74	1	1	448
Elevator	SD	79	204	846	26	62	51	1	1	218
Elevator	SD	81	204	846	26	62	51	1	1	218
Elevator	SD	83	189	636	25	49	38	1	1	90
Elevator	SD	85	190	581	26	48	37	1	1	36
Elevator	SD	87	190	613	26	48	37	1	1	72
Elevator	SD	102	191	604	26	49	38	2	7	46
Elevator	SD	104	218	1057	26	75	64	2	19	329
Elevator	SD	106	231	1295	26	86	75	2	26	483
Elevator	SD	108	106	128	8	10	5	1	1	0

Fig. A.69: Detailed list of response variables for all points of interest (part 8 of 11). The figure provides the specification name, the abstraction criterion, the unique vertex ID and the full list of ASRN and specification measures.

Spec.	Deps.	V-ID	V	A	S/E	P	CC	v'(i)	v'(u)	DU
Elevator	SD	111	192	642	25	49	38	1	1	90
Elevator	SD	113	207	852	26	62	51	1	1	218
Elevator	SD	115	207	852	26	62	51	1	1	218
Elevator	SD	117	109	135	9	11	6	1	1	0
Elevator	SD	119	193	619	26	48	37	1	1	72
Elevator	SD	124	194	610	26	49	38	2	7	46
Elevator	SD	126	221	1063	26	75	64	2	19	329
Elevator	SD	128	234	1300	26	86	75	2	26	482
Elevator	SD	130	109	128	8	10	5	1	1	0
Elevator	SD	133	195	648	25	49	38	1	1	90
Elevator	SD	135	210	858	26	62	51	1	1	218
Elevator	SD	137	210	858	26	62	51	1	1	218
Elevator	SD	139	112	135	9	11	6	1	1	0
Elevator	SD	141	196	625	26	48	37	1	1	72
Elevator	SD	146	197	616	26	49	38	2	7	46
Elevator	SD	148	235	1303	26	86	75	2	26	483
Elevator	SD	150	226	1073	26	75	64	2	19	329
Elevator	SD	152	226	1056	26	75	64	1	1	329
Elevator	SD	154	198	654	25	49	38	1	1	90
Elevator	SD	156	213	864	26	62	51	1	1	218
Elevator	SD	158	213	864	26	62	51	1	1	218
Elevator	SD	160	112	126	8	10	6	1	1	0
Elevator	SD	162	199	631	26	48	37	1	1	72
Elevator	SD	167	200	622	26	49	38	2	7	46
Elevator	SD	169	238	1308	26	86	75	2	26	482
Elevator	SD	171	229	1079	26	75	64	2	19	329
Elevator	SD	173	229	1062	26	75	64	1	1	329
Elevator	SD	175	201	660	25	49	38	1	1	90
Elevator	SD	177	216	870	26	62	51	1	1	218
Elevator	SD	179	216	870	26	62	51	1	1	218
Elevator	SD	181	112	115	7	9	6	1	1	0
Elevator	SD	183	202	637	26	48	37	1	1	72
Elevator	SD	188	203	628	26	49	38	2	7	46
Elevator	SD	190	218	901	26	63	52	2	13	229
Elevator	SD	192	242	1310	26	86	75	2	26	477
Elevator	SD	194	233	1087	26	75	64	2	19	329
Elevator	SD	196	221	896	26	63	52	1	1	229
Elevator	SD	198	205	668	25	49	38	1	1	90
Elevator	SD	200	220	878	26	62	51	1	1	218
Elevator	SD	202	110	101	5	7	5	1	1	0
Elevator	SD	204	220	878	26	62	51	1	1	218
Elevator	SD	206	113	108	6	8	6	1	1	0
Elevator	SD	208	206	645	26	48	37	1	1	72
Elevator	SD	213	207	636	26	49	38	2	7	46
Elevator	SD	215	222	909	26	63	52	2	13	229
Elevator	SD	217	246	1317	26	86	75	2	26	476
Elevator	SD	219	237	1095	26	75	64	2	19	329
Elevator	SD	221	225	904	26	63	52	1	1	229
Elevator	SD	223	209	676	25	49	38	1	1	90
Elevator	SD	225	224	886	26	62	51	1	1	218
Elevator	SD	227	224	886	26	62	51	1	1	218
Elevator	SD	229	114	95	5	7	6	1	1	0
Elevator	SD	231	210	653	26	48	37	1	1	72
Elevator	SD	266	247	1261	26	85	74	1	1	448
Elevator	SD	268	209	676	25	49	38	1	1	90
Elevator	SD	270	224	886	26	62	51	1	1	218
Elevator	SD	272	224	886	26	62	51	1	1	218
Elevator	SD	274	210	621	26	48	37	1	1	36
Elevator	SD	276	121	120	6	10	8	1	1	0
Elevator	SD	281	238	1095	26	75	64	2	7	338
Elevator	SD	283	248	1263	26	85	74	1	1	448
Elevator	SD	285	210	678	25	49	38	1	1	90
Elevator	SD	287	225	888	26	62	51	1	1	218
Elevator	SD	289	225	888	26	62	51	1	1	218

Fig. A.70: Detailed list of response variables for all points of interest (part 9 of 11). The figure provides the specification name, the abstraction criterion, the unique vertex ID and the full list of ASRN and specification measures.

Spec.	Deps.	V-ID	V	A	S/E	P	CC	v'(I)	v'(u)	DU
Elevator	SD	291	211	623	26	48	37	1	1	36
Elevator	SD	293	131	149	9	13	8	1	1	0
Elevator	SD	300	124	98	7	10	5	1	1	0
Elevator	SD	304	122	89	6	9	5	1	1	0
Elevator	SD	310	125	93	6	9	5	1	1	0
Elevator	SD	325	122	73	4	7	5	1	1	0
Elevator	SD	335	116	50	2	5	4	1	1	0
Elevator	SD	342	120	58	3	6	5	1	1	0
Elevator	SD	344	117	50	2	5	4	1	1	0
Mean			**191,167**	**710,216**	**22,029**	**51,588**	**42,157**	**1,245**	**5,069**	**169,951**
Elevator	SC	21	8	20	2	3	3	1	1	0
Elevator	SC	23	10	24	2	3	3	1	1	0
Elevator	SC	25	12	28	2	3	3	1	1	0
Elevator	SC	27	14	32	2	3	3	1	1	0
Elevator	SC	32	15	37	2	3	3	1	1	0
Elevator	SC	34	16	39	2	3	3	1	1	0
Elevator	SC	36	17	41	2	3	3	1	1	0
Elevator	SC	38	18	43	2	3	3	1	1	0
Elevator	SC	40	23	54	3	4	4	1	1	0
Elevator	SC	42	25	58	3	4	4	1	1	0
Elevator	SC	47	36	102	4	10	9	1	1	0
Elevator	SC	54	29	47	4	6	5	1	1	0
Elevator	SC	56	51	154	6	19	18	7	7	1
Elevator	SC	58	52	157	6	19	18	7	7	1
Elevator	SC	60	53	158	6	19	18	7	7	1
Elevator	SC	62	54	160	6	19	18	7	7	1
Elevator	SC	64	57	169	6	21	20	7	7	0
Elevator	SC	66	58	171	6	21	20	7	7	0
Elevator	SC	75	37	49	4	6	5	1	1	0
Elevator	SC	77	58	168	6	19	18	7	7	1
Elevator	SC	79	62	185	6	21	20	7	7	2
Elevator	SC	81	63	187	6	21	20	7	7	2
Elevator	SC	83	62	176	6	19	18	7	7	1
Elevator	SC	85	65	185	6	21	20	7	7	0
Elevator	SC	87	66	187	6	21	20	7	7	0
Elevator	SC	102	61	111	9	11	6	1	1	0
Elevator	SC	104	60	111	8	11	6	1	1	0
Elevator	SC	106	62	118	8	12	7	1	1	0
Elevator	SC	108	135	451	14	50	45	25	25	7
Elevator	SC	111	135	459	14	50	45	25	25	14
Elevator	SC	113	136	454	14	50	45	25	25	8
Elevator	SC	115	137	461	14	50	45	25	25	12
Elevator	SC	117	138	457	14	50	45	25	25	6
Elevator	SC	119	141	460	14	52	47	25	25	0
Elevator	SC	124	85	125	9	11	6	1	1	0
Elevator	SC	126	83	123	8	11	6	1	1	0
Elevator	SC	128	84	128	8	12	7	1	1	0
Elevator	SC	130	140	461	14	50	45	25	25	7
Elevator	SC	133	140	469	14	50	45	25	25	14
Elevator	SC	135	141	469	14	50	45	25	25	12
Elevator	SC	137	142	466	14	50	45	25	25	8
Elevator	SC	139	143	467	14	50	45	25	25	6
Elevator	SC	141	146	470	14	52	47	25	25	0
Elevator	SC	146	87	114	8	10	6	1	1	0
Elevator	SC	148	86	117	7	11	7	1	1	0
Elevator	SC	150	85	112	7	10	6	1	1	0
Elevator	SC	152	145	473	14	50	45	25	25	7
Elevator	SC	154	146	481	14	50	45	25	25	14
Elevator	SC	156	147	476	14	50	45	25	25	8
Elevator	SC	158	148	483	14	50	45	25	25	12
Elevator	SC	160	149	479	14	50	45	25	25	6
Elevator	SC	162	152	482	14	52	47	25	25	0
Elevator	SC	167	90	103	7	9	6	1	1	0

Fig. A.71: Detailed list of response variables for all points of interest (part 10 of 11). The figure provides the specification name, the abstraction criterion, the unique vertex ID and the full list of ASRN and specification measures.

Spec.	Deps.	V-ID	V	A	S/E	P	CC	v'(i)	v'(u)	DU
Elevator	SC	169	89	106	6	10	7	1	1	0
Elevator	SC	171	88	101	6	9	6	1	1	0
Elevator	SC	173	151	485	14	50	45	25	25	7
Elevator	SC	175	152	493	14	50	45	25	25	14
Elevator	SC	177	153	493	14	50	45	25	25	12
Elevator	SC	179	154	490	14	50	45	25	25	8
Elevator	SC	181	155	491	14	50	45	25	25	6
Elevator	SC	183	158	494	14	52	47	25	25	0
Elevator	SC	188	93	94	6	8	6	1	1	0
Elevator	SC	190	90	87	5	7	5	1	1	0
Elevator	SC	192	91	92	5	8	6	1	1	0
Elevator	SC	194	91	92	5	8	6	1	1	0
Elevator	SC	196	157	496	14	50	45	25	25	7
Elevator	SC	198	158	501	14	50	45	25	25	11
Elevator	SC	200	159	500	14	50	45	25	25	8
Elevator	SC	202	160	501	14	50	45	25	25	7
Elevator	SC	204	161	504	14	50	45	25	25	8
Elevator	SC	206	162	505	14	50	45	25	25	6
Elevator	SC	208	165	508	14	52	47	25	25	0
Elevator	SC	213	97	83	5	7	6	1	1	0
Elevator	SC	215	94	76	4	6	5	1	1	0
Elevator	SC	217	95	81	4	7	6	1	1	0
Elevator	SC	219	95	81	4	7	6	1	1	0
Elevator	SC	221	164	510	14	50	45	25	25	7
Elevator	SC	223	165	515	14	50	45	25	25	11
Elevator	SC	225	166	514	14	50	45	25	25	8
Elevator	SC	227	167	521	14	50	45	25	25	12
Elevator	SC	229	168	517	14	50	45	25	25	6
Elevator	SC	231	171	520	14	52	47	25	25	0
Elevator	SC	266	159	276	14	31	24	10	10	1
Elevator	SC	268	160	279	14	31	24	10	10	2
Elevator	SC	270	161	281	14	31	24	10	10	2
Elevator	SC	272	162	283	14	31	24	10	10	2
Elevator	SC	274	165	291	14	33	26	10	10	0
Elevator	SC	276	166	294	14	33	26	10	10	0
Elevator	SC	281	132	148	8	14	9	1	1	0
Elevator	SC	283	160	261	13	29	22	10	10	1
Elevator	SC	285	161	264	13	29	22	10	10	2
Elevator	SC	287	162	266	13	29	22	10	10	2
Elevator	SC	289	163	268	13	29	22	10	10	2
Elevator	SC	291	166	276	13	31	24	10	10	0
Elevator	SC	293	167	279	13	31	24	10	10	0
Elevator	SC	300	131	104	7	10	5	1	1	0
Elevator	SC	304	128	93	6	9	5	1	1	0
Elevator	SC	310	128	95	6	9	5	1	1	0
Elevator	SC	325	122	73	4	7	5	1	1	0
Elevator	SC	335	116	50	2	5	4	1	1	0
Elevator	SC	342	120	58	3	6	5	1	1	0
Elevator	SC	344	117	50	2	5	4	1	1	0
			110,000	**261,284**	**9,343**	**27,196**	**23,569**	**11,471**	**11,471**	**2,971**

Fig. A.72: Detailed list of response variables for all points of interest (part 11 of 11). The figure provides the specification name, the abstraction criterion, the unique vertex ID and the full list of ASRN and specification measures.

B. FURTHER READINGS

B.1 Complexity of Specifications

- *Introduction to Formal Methods.* There are several books introducing formal methods. One can gain a good overview of different formal notations in Alagar's and Periyasamy's book (Specification of Software Systems) [AP98]. The book covers logic and algebraic specifications as well as Z, VDM and Larch. A good motivation for using formal methods can be found in Jonathan Jacky's book (The Way of Z) [Jac96] - nevertheless, one of the best books introducing Z is that of Anthony Diller (Z – An Introduction to Formal Methods) [Dil99] and an article from Spivey (An introduction to Z and formal specifications) [Spi89a]. More insight into refinement and proof can be found in Jim Woodcock's and Jim Davis' book (Using Z) [WD96] and in the book of Cliff Jones (Systematic Software Development using VDM) [Jon90]. The Formal Methods Group in Europe (FME) provides a home-page[1] with a list of several books and articles – a starting point for the interested reader might be the technical report of Edmund Clarke and Jeannette Wing [CW96].

- *Arguments for using Formal Methods.* All the books mentioned above provide some kind of motivation chapters. There is much data about the quoted CDIS project, and an article of Pfleeger and Hatton [PH97] summarizes the lessons learned during that project. Once again, there are many articles that are cited on the FME home-page. One notable article from Hamilton, Covington and Kelly [HCK+95] summarizes trial projects conducted together with the NASA.

- *Controversy.* Probably the most-widely known article dealing with controversial aspects of formal methods is that of Anthony Hall (Seven Myths of Formal Methods) [Hal90]. His arguments are extended by Bowen and Hinchey in 1995 (Seven More Myths of Formal Methods) [BH95] - especially the lack of suitable tools and missing connections to other representational forms are elaborated. A thorough collection of articles dealing with the question whether and when formal methods pay-off can be found in the April 1996 edition of IEEE Computer [HDR+96]. Anthony Hall, David Dill, Michael Holloway, Ricky Butler

[1] http://www.fmeurope.org (last visited: September 2003)

and Pamela Zave provide an insight (and lessons learned) into the use of formal methods in projects conducted at Praxis, SRI International, NASA Langley Research Center and AT&T.

- *Quality and complexity.* A sound introduction to the field of software metrics can be found in the Software Engineering Institute Curriculum Module SEI-CM-12-1.1 [Mil88]. It summarizes different product metrics for size and complexity and also discusses process metrics. It furthermore provides a richly annotated list of relevant bibliography. A more philosophical view on the measure of complexity can be found in the PhD thesis of Bruce Edmonds [Edm99]. As there are different formulations of complexity across different fields of study, this imprecision is sorted out and a theoretical framework for the formalizations of complexity is provided. A more recent status report on Software Measurement (and a more controversial view) can be found in the article of Pfleeger, Jeffery, Curtis and Kitchenham in the March/April 1997 edition of IEEE Software [PJCK97].

B.2 Specification Abstractions

- *State-of-the-Art.* In general specification languages are not executable. This means that there are no compilers, linkers or tools which one would expect from programming languages. However, approaches have been developed in order to animate and execute at least subsets of specifications. Diller [Dil99, p.271ff] describes how to convert Z specifications to Miranda, a functional programming language designed by David Turner [Tur86]. Diller describes the steps necessary to transform a Z specification to a Miranda animation by writing a simple interactive program. The VDM toolkit environment provides a wide range of tools for animation and automatic transformation to another programming language (C++ or Java) [LL91, FL96]. A compact overview can be found in the technical paper of Hazel, Strooper and Traynor [HST98].

- *Components of Specifications.* There are not many articles dealing with the identification of components of specifications. The article of Chang and Richardson [CR94] presents an approach for the identification of static and dynamic specification slices. However, a dynamic slice is nothing else than an static slice which is applied after eliminating specific operation schemata (those which are not needed during the "execution" of the specification). Modules are identified in the paper of Carrington et.al [CDHW93] by looking both at a cross-reference table and at the use of state variables in different operation schemata. Up to now there are no clustering algorithms for specification. However, a thorough introduction to clustering can be found in the paper of Wiggerts [Wig97]. He

defines similarity measures between abstract entities and provides four clustering algorithms.

B.3 Specification's Complexity

- *Measuring Complexity.* There are quite a lot of articles which are concerned with various types of measures. A rigorous and practical approach is described in the book of Fenton and Pfleeger [FP97]. Jones discusses the unit-of-measure situation in his 1978 article [Jon78] which is based on the LOC approach. He explains how to examine program quality and presents rules of the thumb for cost and effort estimation. An excellent article of McCabe and Butler [McC89] introduces McCabes metrics for programs and for design documents based on flow-graphs. There is an excellent article published at the SEI in 1988 by Everald E. Mills [Mil88]. He introduces software metrics and provides an annotated list of over 70 references to the most important articles. The topic of cohesion is thoroughly introduced by Arun Lakhotia [Lak97]. He provides an introduction to the topic of cohesion, presents some case studies and compares the result to other approaches to be found in literature.

C. SPECIFICATIONS IN USE

C.1 Birthday Book

The birthday book specification [Spi89b] is used in this work extensively. In the subsequent section the Z specification of the birthday book and the transformation to the eSRN are presented. The transformation rules are presented in Chap. 5.3.3.

C.1.1 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 *Success* 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

$$\Delta BB$$
$$name? : NAME$$
$$date? : DATE$$
$$\hline$$
$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

Delete

$$\Delta BB$$
$$name? : NAME$$
$$\hline$$
$$name? \in known$$
$$birthday' = birthday \setminus \{name? \mapsto birthday(name?)\}$$

To indicate the success of an operation the result OK is returned.

Success

$$result! : REPORT$$
$$\hline$$
$$result! = OK$$

With the above operation schemata the functioning system consists of successfully performed add or delete operations.

$$FunctioningDB == (Add \land Success) \lor (Delete \land Success)$$

C.1.2 Birthday Book eSRN Transformation

The following two figures demonstrate the mapping between specification primes and vertices in the eSRN representation.

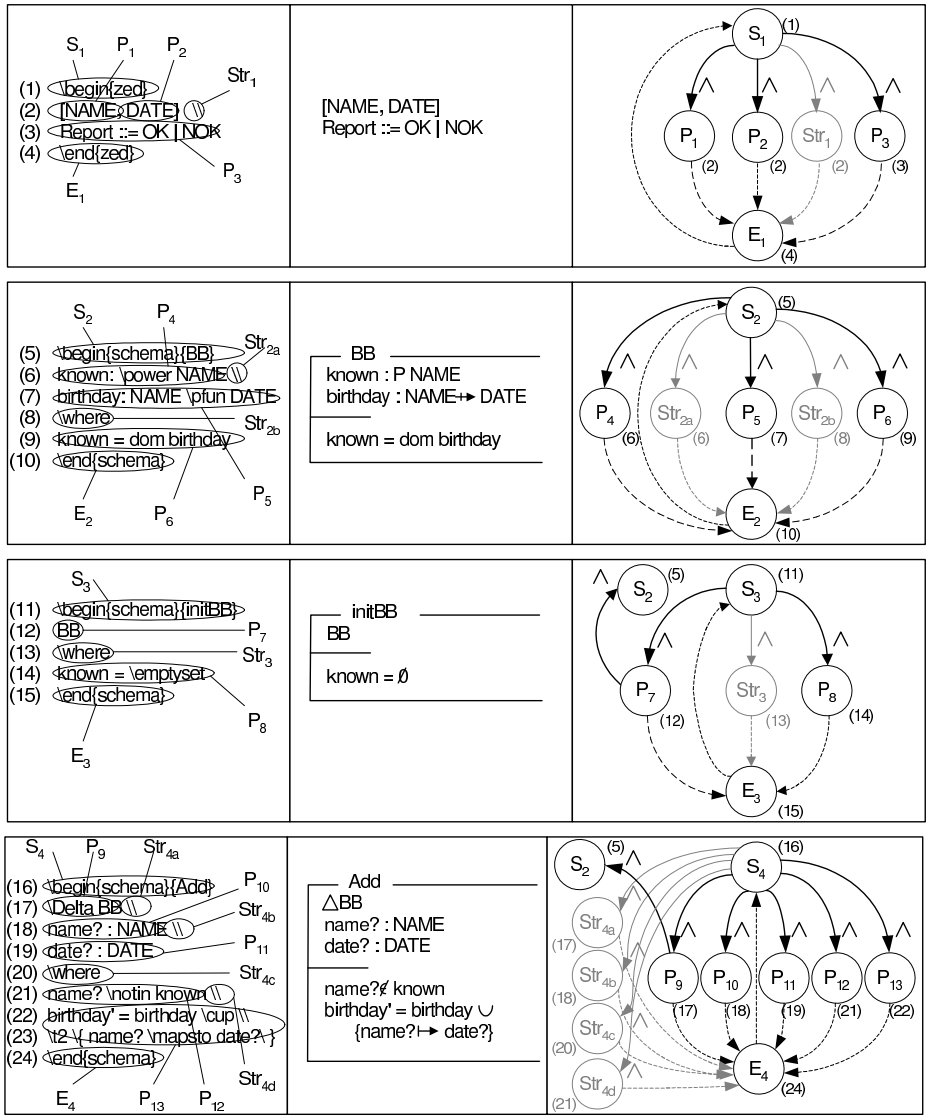


Fig. C.1: Transformation of the first part of the birthday book specification to the eSRN. Prime and structural vertices are labelled by rounded boxes to ease mapping between the specification source and the net.

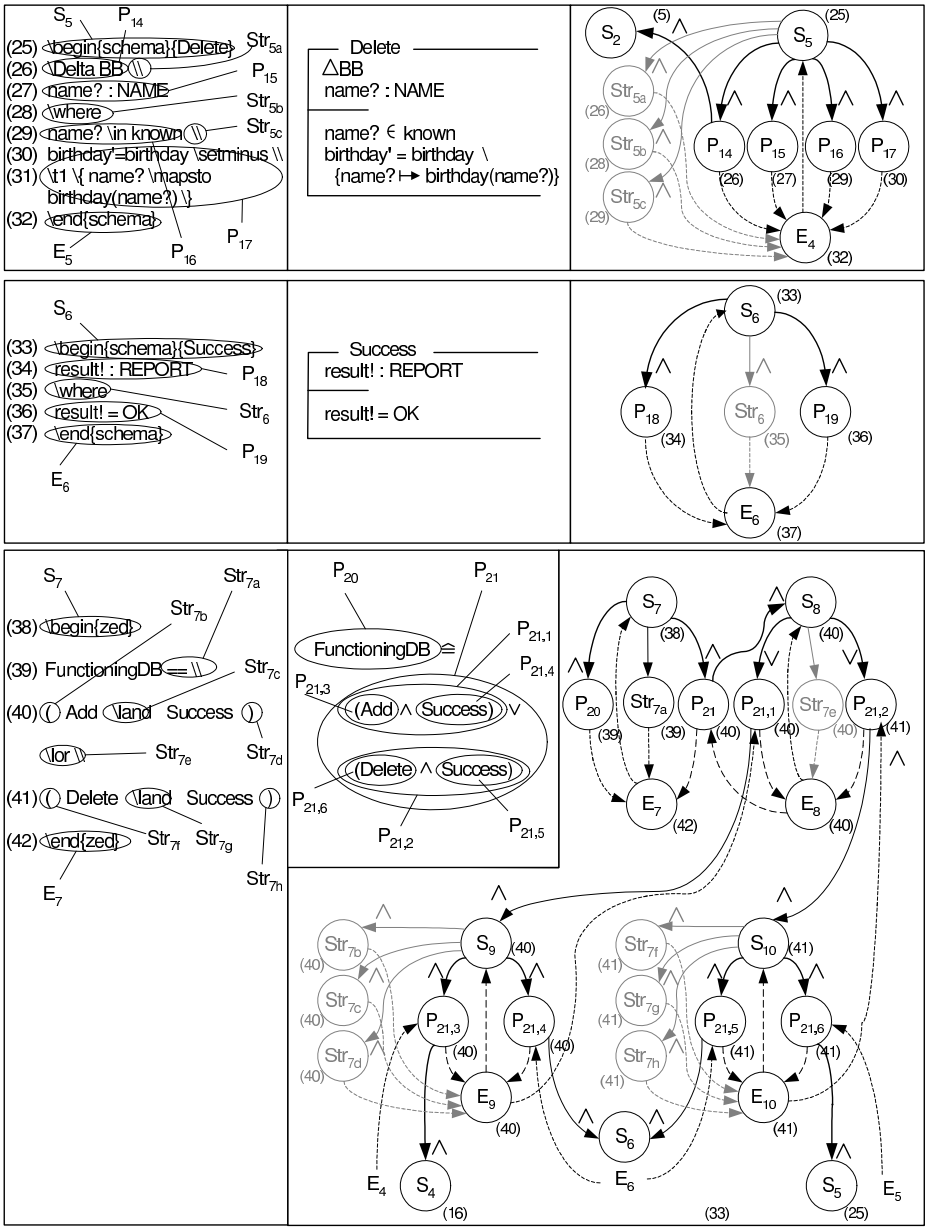


Fig. C.2: Transformation of the second part of the birthday book specification to an eSRN. Prime and structural vertices are labelled by rounded boxes to ease mapping between the specification source and the net.

C.2 Petrol Station

This specification is a simple solution to the petrol station problem. It is a small system to be written by students during the practical classes of a lecture called "Specification and Verification" at the University of Klagenfurt.

The system consists of several pump stations (GS) and vehicles (VH).

$$[GS, VH]$$

The state space consists of a set of pump stations. Every pump station has its own queue of vehicles and an operating position.

PetrolStation

$$petrol : \mathbb{P}\, GS$$
$$waiting : GS \pfun \operatorname{seq} VH$$
$$operation : GS \pfun VH$$

$$\operatorname{dom} waiting \subseteq petrol$$
$$\operatorname{dom} operation \subseteq petrol$$
$$\forall z1, z2 : GS \mid z1 \in petrol \land z2 \in petrol \land z1 \neq z2 \bullet \operatorname{ran}(waiting\ z1) \cap \operatorname{ran}(waiting\ z2) = \langle\,\rangle \land operation\ z1 \neq operation\ z2$$
$$\forall fz : VH;\ z : GS \mid z \in petrol \land fz \in \operatorname{ran} operation \bullet fz \notin \operatorname{ran}(waiting\ z)$$

At the beginning there are no pump stations; thus, no vehicles are waiting for refuelling. (Please note that operations for adding pump stations to the petrol station are missing. For the scope of the practical class these operations have been neglected.)

InitPetrolStation

$$PetrolStation$$

$$petrol = \varnothing$$
$$waiting = \varnothing$$
$$operation = \varnothing$$

It is possible that there is no vehicle in the queue; thus, one can arrive and start refuelling.

ArrivalAndRefuel

$\Delta PetrolStation$
$fz? : VH$
$z? : GS$

$z? \in petrol$
$z? \notin \text{dom } operation$
$fz? \notin \text{ran } operation$
$\forall GS : GS \mid GS \in petrol \bullet fz? \notin \text{ran}(waiting\ GS)$
$operation' = operation \oplus \{z? \mapsto fz?\}$
$waiting' = waiting$
$petrol' = petrol$

On the other hand if there are vehicles in the queue, one has to wait.

ArriveAndWait

$\Delta Tankstelle$
$fz? : VH$
$z? : GS$

$z? \in petrol$
$\#(\text{ran } operation) = \#petrol$
$fz? \notin \text{ran } operation$
$\forall GS : GS \mid GS \in petrol \bullet fz? \notin \text{ran}(waiting\ GS)$
$operation' = operation$
$waiting' = waiting \oplus \{z? \mapsto (waiting\ z?) \frown \langle fz? \rangle\}$
$petrol' = petrol$

There are two possibilities when arriving: the queue is either empty and there is no need for waiting or there are vehicles in the queue, and one has to wait.

$Arrival \mathrel{\widehat=} ArriveAndFuel \lor ArriveAndWait$

After refuelling the vehicle should leave the petrol station. If there is another vehicle in the queue it can move up. Otherwise the vehicle leaves the petrol station and the queue stays empty.

LeaveEmptyQueue

$$
\begin{array}{l}
\Delta PetrolStation \\
fz? : VH \\
z : GS \\
\hline
\exists_1\, z : GS \mid operation\, z = fz? \land waiting\, z = \langle\,\rangle \bullet \\
\quad operation' = operation \setminus \{z \mapsto fz?\} \land \\
\quad waiting' = waiting \land \\
\quad petrol' = petrol
\end{array}
$$

LeaveNonEmptyQueue

$$
\begin{array}{l}
\Delta PetrolStation \\
fz? : VH \\
z : GS \\
\hline
\exists_1\, z : GS \mid operation\, z = fz? \land waiting\, z \neq \langle\,\rangle \bullet \\
\quad operation' = operation \oplus \{z \mapsto head(waiting\, z)\} \land \\
\quad waiting' = waiting \oplus \{z \mapsto tail(waiting\, z)\} \land \\
\quad petrol' = petrol
\end{array}
$$

It is also possible that a vehicle leaves the petrol station without refuelling.

LeaveWithoutFuel

$$
\begin{array}{l}
\Delta PetrolStation \\
fz? : VH \\
\hline
\exists\, z : GS;\ i : \mathbb{Z} \mid z \in petrol \land (i \mapsto fz?) \in waiting\, z \bullet \\
\quad waiting' = waiting \oplus \{z \mapsto squash((waiting\, z) \setminus \{i \mapsto fz?\}\,)\,\} \land \\
\quad operation' = operation \land \\
\quad petrol' = petrol
\end{array}
$$

$$Leave \mathrel{\widehat=} LeaveEmptyQueue \lor LeaveNonEmptyQueue \lor LeaveWithoutFuel$$

$$PetrolStation \mathrel{\widehat=} InitPetrolStation \land (\, Arrive \lor Leave\,)$$

C.3 Elevator Specification

The Elevator specification [CR94] describes a simple system consisting of one elevator. The elevator reacts to button presses 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 (*ElevatorButtonEvent*). Secondly, a passenger calls for the elevator (*FloorButtonEvent*).

ElevatorButtonEvent

$$
\begin{array}{l}
\Delta Elevator \\
Floor? : \mathbb{N}_1 \\
\hline
Floor? \leq MaxFloor \\
CurrentFloor' = CurrentFloor \\
Requests' = Requests \cup \{Floor?\} \\
UpCalls' = UpCalls \\
DownCalls' = DownCalls \\
Dir' = Dir \\
Door' = Door
\end{array}
$$

FloorButtonEvent

$$
\begin{array}{l}
\Delta Elevator \\
Floor? : \mathbb{N}_1 \\
CallDir? : Direction \\
\hline
Floor? \leq MaxFloor \\
CurrentFloor' = CurrentFloor \\
(CallDir? = down) \Rightarrow (UpCalls' = UpCalls) \land \\
\quad (DownCalls' = DownCalls \cup \{Floor?\}) \\
(CallDir? = up) \Rightarrow (DownCalls' = DownCalls) \land \\
\quad (UpCalls' = UpCalls \cup \{Floor?\}) \\
Requests' = Requests \\
Dir' = Dir \\
Door' = Door
\end{array}
$$

$$
\begin{array}{l}
PassengerEvent == ElevatorButtonEvent \\
\qquad \lor \; FloorButtonEvent
\end{array}
$$

The movement of the elevator is complex. An elevator might move up or down (*BasicMoveUp* or *BasicMoveDown*) when it is servicing requests or calls.

BasicMoveUp

$$
\begin{array}{l}
\Delta Elevator \\
\hline
Dir = up \\
Requests \cup UpCalls \neq \varnothing \\
CurrentFloor \leq max\ (Requests \cup UpCalls) \\
CurrentFloor' = min\ \{x : \mathbb{N}_1 \mid x \in \\
\qquad (Requests \cup UpCalls) \\
\qquad \land\ x > CurrentFloor\} \\
Requests' = Requests \setminus \{CurrentFloor'\} \\
UpCalls' = UpCalls \\
DownCalls' = DownCalls \setminus \{CurrentFloor'\} \\
Dir' = up \\
Door' = Door
\end{array}
$$

BasicMoveDown

$$
\begin{array}{l}
\Delta Elevator \\
\hline
Dir = down \\
Requests \cup DownCalls \neq \varnothing \\
CurrentFloor \leq min\ (Requests \cup DownCalls) \\
CurrentFloor' = max\ \{x : \mathbb{N}_1 \mid x \in \\
\qquad (Requests \cup DownCalls) \\
\qquad \land\ x < CurrentFloor\} \\
Requests' = Requests \setminus \{CurrentFloor'\} \\
UpCalls' = UpCalls \setminus \{CurrentFloor'\} \\
DownCalls' = DownCalls \\
Dir' = down \\
Door' = Door
\end{array}
$$

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*).

ChangeUpToDown

$\Delta Elevator$

$Dir = up$
$(Requests \cup UpCalls \neq \varnothing \land$
$CurrentFloor > max\ (Requests \cup UpCalls)) \lor$
$Requests \cup UpCalls = \varnothing$
$Requests \cup DownCalls \neq \varnothing$
$CurrentFloor' = max\ (Requests \cup DownCalls)$
$Requests' = Requests \setminus \{CurrentFloor'\}$
$UpCalls' = UpCalls$
$DownCalls' = DownCalls \setminus \{CurrentFloor'\}$
$Dir' = down$
$Door' = Door$

ChangeDownToUp

$\Delta Elevator$

$Dir = down$
$(Requests \cup DownCalls \neq \varnothing \land$
$CurrentFloor <$
$\quad min\ (Requests \cup DownCalls)) \lor$
$Requests \cup DownCalls = \varnothing$
$Requests \cup UpCalls \neq \varnothing$
$CurrentFloor' = min\ (Requests \cup UpCalls)$
$Requests' = Requests \setminus \{CurrentFloor'\}$
$UpCalls' = UpCalls \setminus \{CurrentFloor'\}$
$DownCalls' = DownCalls$
$Dir' = up$
$Door' = Door$

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

$$\Delta Elevator$$

$$Dir = up$$
$$UpCalls \neq \varnothing$$
$$CurrentFloor > max\ UpCalls$$
$$DownCalls \cup Requests = \varnothing$$
$$CurrentFloor' = min\ UpCalls$$
$$Requests' = Requests$$
$$UpCalls' = UpCalls \setminus \{CurrentFloor'\}$$
$$DownCalls' = DownCalls$$
$$Dir' = up$$
$$Door' = Door$$

RestartMovingDown

$$\Delta Elevator$$

$$Dir = down$$
$$DownCalls \neq \varnothing$$
$$CurrentFloor < min\ DownCalls$$
$$UpCalls \cup Requests = \varnothing$$
$$CurrentFloor' = max\ DownCalls$$
$$Requests' = Requests$$
$$UpCalls' = UpCalls$$
$$DownCalls' = DownCalls \setminus \{CurrentFloor'\}$$
$$Dir' = down$$
$$Door' = Door$$

$$Move == BasicMoveUp \lor BasicMoveDown \lor ChangeUpToDown \lor ChangeDownToUp \lor RestartMovingUp \lor RestartMovingDown$$

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.

$$MoveCycle == (CloseDoor \fatsemi PassengerEvent \fatsemi Move \fatsemi (PassengerEvent) \fatsemi \\ OpenDoor \fatsemi PassengerEvent)^*$$
$$ElevatorCycle == (MoveCycle \lor NoRequestsOrCalls)^*$$

$$FunctioningElevator == (PassengerEvent \fatsemi ElevatorCycle)^*$$

C.4 ITC Window Manager

The "ITC Window Manager"-specification describes one part of the "Andrew" distributed system which is a window manager developed at the Information Technology Center (ITC) at Carnegie-Mellon University (CMU). In the following there is a shortened description of the specification. It follows the description provided in [Bow96, p.169ff].

The display is made up of a set of pixels with positions defined in XY coordinate space. The display is a fixed size bounded rectangle in the XY plane.

$$Xsize : \mathbb{N}_1$$
$$Ysize : \mathbb{N}_1$$

$$Xrange == 0 \ldots (Xsize - 1)$$

$$Yrange == 0 \ldots (Ysize - 1)$$

$$Pixel == (Xrange \times Yrange)$$

Many operations are applied to pairs of pixels.

PixelPair

$$pix_1 : Pixel$$
$$pix_2 : Pixel$$
$$x_1 : Xrange$$
$$x_2 : Xrange$$
$$y_1 : Yrange$$
$$y_2 : Yrange$$

$$pix_1 = (x_1, y_1)$$
$$pix_2 = (x_2, y_2)$$

A display contains a number of bit-planes. This may be considered as the Z direction of the display.

$$Zsize : \mathbb{N}_1$$

There is one of two values (cleared or set) for each bit in a bit-plane.

$$ClearVal == 0$$

$$SetVal == 1$$

$$BitVal == \{ClearVal, SetVal\}$$

The value of a pixel at a particular position is modelled as a function from a bit-plane number to a bit value.

$$Zrange == 0 \ldots (Zsize - 1)$$

$$Value == (Zrange \fun BitVal)$$

If all the bits are clear the "Value" is considered "Black". If all the bits are set it is considered "White".

$$Black == (\mu\, val : Value \mid \ran val = \{ClearVal\})$$

$$White == (\mu\, val : Value \mid \ran val = \{SetVal\})$$

A pixel map consists of a (partial) function from pixel positions to the value of the pixel contents. This is used to describe part of a display, such as a window.

$$Pixmap == (Pixel \pfun Value)$$

In order to set all the range of a pixel map to a particular value, a function to set the range of a relation to a particular value is provided.

$$\begin{array}{|l} setval : V \fun P \pfun V \fun P \pfun V \\ \hline \forall\, v : V;\ p : (P \pfun V) \bullet setval\ v\ p = \\ \qquad (\mu\, m : P \pfun V \mid (\dom m = \dom p \land \ran m = \{v\})) \end{array}$$

A window might contain text fields. This text, which can also be empty, is denoted as "String".

$$[String]$$

Each window has a number of pieces of information associated with it. A header area is used for titles and other information, a separate body area holds the actual contents of the window. These do not overlap and together they make up the pixel map of the displayed window.

Map

$$
\begin{array}{l}
header : Pixmap \\
body : Pixmap \\
map : Pixmap \\
area : \mathbb{P}\, Pixel \\
\hline
\langle header, body \rangle \text{ partition } map \\
area = \text{dom } map
\end{array}
$$

The user can request a window to lie within a specified range of dimensions and can also explicitly ask for a window body to be hidden from view or exposed on the screen. Each window has a title which can be set by the user. This information is used by the window manager to lay out the window on the screen.

$$HideExpose ::= Hide \mid Expose$$

Control

$$
\begin{array}{l}
title : String \\
control : HideExpose \\
xylimits : Pixel \times Pixel \\
\hline
(first\ xylimits) \leq (second\ xylimits)
\end{array}
$$

Map and Control make up the description of the particular window.

$$Info == Map \wedge Control$$

There are a finite number of windows on a particular screen. One of these is considered to be the currently selected window. It might be undefined, too. Most WM library functions take effect on the currently selected window. Each window has information, including a pixel map, associated with it.

$$[Window]$$

$$\mid Undefined : Window$$

WMZero

$$
\begin{array}{l}
windows : \mathbb{F}\ Window \\
current : Window \\
contents : Window \pfun Info \\
\hline
Undefined \notin windows \\
windows = \mathrm{dom}\ contents \\
current \in windows \cup \{ Undefined \}
\end{array}
$$

The display screen consists of the background overlaid with windows. The window pixel maps do not overlap. All windows are contained within the background area.

WMOne

$$
\begin{array}{l}
WMZero \\
maps : Window \pfun Pixmap \\
areas : Window \pfun (\mathbb{P}\ Pixel) \\
screen : Pixmap \\
background : Pixmap \\
\hline
maps = contents \fatsemi (\lambda\ Info \bullet map) \\
areas = contents \fatsemi (\lambda\ Info \bullet area) \\
\mathrm{disjoint}\ areas \\
\bigcup(\mathrm{ran}\ areas) \subseteq \mathrm{dom}\ background \\
screen = background \oplus \bigcup(\mathrm{ran}\ maps)
\end{array}
$$

The WM process can handle at most 20 windows, including hidden windows and windows requested by other programs, at one time.

$$
\begin{array}{l}
MaxWindows : \mathbb{N} \\
\hline
MaxWindows = 20
\end{array}
$$

This limitation is included in the model of the state.

WMTwo

$$
\begin{array}{l}
WMOne \\
\hline
\sharp windows \leq MaxWindows
\end{array}
$$

The size of a window on the users display is one of the resources that the Window Manager allocates. A program can request a given size, and WM will take the requested size into account when making decisions, but it does not guarantee a particular size. This process is modelled as a function of the system. The number of windows is not changed by this function. Additionally, control information supplied by the user is left unchanged.

$$WINDOWS : Window \pfun Info$$

WM

$$WMTwo$$
$$adjust : WINDOWS \fun WINDOWS$$
$$\forall w, w' : WINDOWS \mid w' = adjust(w) \bullet$$
$$\qquad \sharp w' = \sharp w \land$$
$$\qquad w' \fatsemi (\lambda Info \bullet \Theta Control) = w \fatsemi (\lambda Info \bullet \Theta Control)$$

Initially there are no windows and the current window is undefined.

InitWM

$$WM'$$
$$windows' = \varnothing$$
$$current' = undefined$$

Many operations are concerned with the current window. Hence a schema giving a partial specification covering all common aspects of such operations is provided. It is used to reduce repetition. A *Phi* is put first to the names to distinguish these from actual operations.

PhiCurrent

$$\Delta WM$$
$$\Delta Info$$
$$current \in windows$$
$$current' = current$$
$$\Theta Info = contents(current)$$
$$contents' = adjust(contents \oplus \{current \mapsto \Theta Info'\})$$

When a window is created, the system adjusts all the windows in the system appropriately. The window body is exposed when it is created.

NewWindow

$$\Delta WM$$
$$w! : Window$$
$$Info$$

$$\sharp windows < MaxWindows$$
$$w! \notin windows \cup \{Undefined\}$$
$$current' = w!$$
$$control = Expose$$
$$contents' = adjust(contents \cup \{w! \mapsto \Theta Info\})$$

The currently selected window can also be deleted.

DeleteWindow

$$\Delta WM$$

$$current \in windows$$
$$current' = Undefined$$
$$contents' = adjust(\{current\} \lhd contents)$$

A program can request a given size range, and WM will take the requested size into account when making decisions, but it does not guarantee a particular size. The rest of the window information is unaffected. The windows will be adjusted by the system as necessary.

SetDimensions

$$PhiCurrent$$
$$minxy? : Pixel$$
$$maxxy? : Pixel$$

$$map' = map$$
$$title' = title$$
$$control' = control$$
$$xylimits' = (minxy?, maxxy?)$$

The size of the body of the currently selected window can be returned. If the window is actually hidden (i.e., WM has adjusted the window to display the header only), then the returned size is empty.

GetDimensions

$$
\begin{array}{l}
PhiCurrent \\
wh! : Pixel \\
xy1 : Pixel \\
xy2 : Pixel \\
\hline
\Theta Info' = \Theta Info \\
\mathrm{dom}\ body = xy1 \ldots xy2 \\
wh! = xy2 - xy1
\end{array}
$$

A window is considered "visible" when both its header and its body are displayed and "hidden" when only its header is displayed. A visible window may be hidden, exposed or selected.

HideMe

$$
\begin{array}{l}
PhiCurrent \\
\hline
map' = map \\
title' = title \\
control' = Hide \\
xylimits' = xylimits
\end{array}
$$

ExposeMe

$$
\begin{array}{l}
PhiCurrent \\
\hline
map' = map \\
title' = title \\
control' = Expose \\
xylimits' = xylimits
\end{array}
$$

SelectWindow

$$
\begin{array}{l}
\Delta WM \\
w? : Window \\
\hline
w? \in windows \\
current' = w? \\
contents' = contents
\end{array}
$$

The title of a window may be set. This involves placing a text string in the header section of the window contents.

SetTitle

$$PhiCurrent$$
$$s? : String$$

$$map' = map$$
$$title' = s?$$
$$control' = control$$
$$xylimits' = xylimits$$

The body of the currently selected window may be set to white.

ClearWindow

$$\Delta PhiCurrent$$

$$header' = header$$
$$body' = setval\ White\ body$$
$$title' = title$$
$$control' = control$$
$$xylimits' = xylimits$$

There is a Null window identifier which is never a valid window.

$$Null : Window$$

WMErr

$$WM$$

$$Null \notin windows$$

Some operations return a window identifier. If this is non-null then the operation is successful.

SuccessWM

$$\Delta WMErr$$
$$w! : Window$$

$$w! \neq Null$$

Alternatively a error may occur. There is a limit on the number of windows which WM can handle. This could cause an error when creating a new window.

TooManyWindows

$$\Xi WMErr$$
$$w! : Window$$
$$\sharp windows \geq MaxWindows$$
$$w! = Null$$

The operation to create a new window is now made total.

$$NewWindow1 == (NewWindow \land SuccessWM) \lor TooManyWindows$$

There are several operations returning an error.

NoCurrentWindow

$$\Xi WMErr$$
$$current = Undefined$$

$$DeleteWindow1 == DeleteWindow \lor NoCurrentWindow$$

$$SetDimensions1 == SetDimensions \lor NoCurrentWindow$$

$$GetDimensions1 == GetDimensions \lor NoCurrentWindow$$

$$SetTitle1 == SetTitle \lor NoCurrentWindow$$

$$ClearWindow1 == ClearWindow \lor NoCurrentWindow$$

InvalidWindow

$$\Xi WMErr$$
$$w? : Window$$
$$w? \notin windows$$

$SelectWindow1 == SelectWindow \lor InvalidWindow$

In the Andrew system there are many window managers, each running on a host workstation on a large network. Some hosts are running WM. All workstations have unique host names and all windows have unique identifiers across the network.

ITC

$$hosts : \mathbb{P}\, String$$
$$wms : String \pfun WM$$
$$\dom wms \subseteq hosts$$
$$\mathrm{disjoint}(wms \fatsemi (\lambda\, WM \bullet windows))$$

InitITC

$$ITC'$$
$$hosts' = \varnothing$$

Hosts can be added or removed. The name cannot be the empty string.

$[EMPTYSTRING]$

AddHost

$$\Delta ITC$$
$$host? : String$$
$$host? \notin hosts \cup EMPTYSTRING$$
$$hosts' = hosts \cup \{host?\}$$
$$wms' = wms$$

RemoveHost

$$\Delta ITC$$
$$host? : String$$
$$host? in hosts$$
$$hosts' = hosts \setminus \{host?\}$$
$$wms' = wms$$

Operations can be initiated on a particular "local" host. These do not affect the host names on the network.

PhiHost

$$\Delta ITC$$
$$localhost : String$$

$$hosts' = hosts$$
$$localhosts \in hosts$$

For example, WM may be executed on a host, and may be subsequently killed.

ExecWM

$$PhiHost$$
$$initwm : WM$$

$$localhost \notin \operatorname{dom} wms$$
$$wms' = wms \cup \{localhost \mapsto initwm\}$$

KillWM

$$PhiHost$$

$$localhost \in \operatorname{dom} wms$$
$$wms' = \{localhost\} \lhd wms$$

WM operations can be modelled in the global context of the network by updating the state of WM on a particular local host which is already running WM.

PhiWM

$$PhiHost$$
$$\Delta WM$$
$$host : String$$

$$\Theta WM = wms\ host$$
$$\Theta WM' = wms'\ host$$

The Window Manager on the local host can be requested to create new windows on any machine on the ITC network that is running a WM process by supplying the appropriate host name. Alternatively, specifying a null host parameter results in a request for a window on the local machine. This is the normal mode of operation.

NewWindowITC

$$
\begin{array}{l}
NewWindow1 \\
PhiWM \\
host? : String \\
\hline
host? = EMPTYSTRING \Rightarrow host = localhost \\
host? \neq EMPTYSTRING \Rightarrow host = host?
\end{array}
$$

Finally an ITC window can be deleted.

$$DeleteWindowITC == DeleteWindow1 \land PhiWM$$

There are several simplifications in this specification. However, the interested reader is referred to [BH95] for other specifications describing window manager systems.

D. GLOSSARY

CC - Conceptual Complexity

The *conceptual complexity CC* represents one factor contributing to the overall complexity of specifications. *CC* is identified by counting the number of primes in the specifcation.

Chunk

Following the definitions in program comprehension [You96a], chunks are syntactic or semantic abstractions of text structures within the source code. Those chunks can be collected and abstracted further, in order to build higher level chunks.

1. *Program Chunk.* According to [BRS+97] a chunk is a (a) *sequence of software instructions that achieves a coherent purpose and that can be understood outside of the context in which it is used.* In the same paper a second definition is given: (b) *A chunk is either a prime, including all primes contained within it, or a sequence of primes that exist within the same programming scope, where for each pair of primes either one prime is data dependent on the other or both primes are data dependent on a third prime within the sequence.* A prime is a fundamental unit, from which structured programs can be built.
2. *Specification Chunk.* A specification chunk is a specification fragment that achieves a coherent purpose and that can be understood outside of the context in which it is used. 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 data-dependent on the other or both primes are data-dependent on a third prime (within the set of primes).

Cluster

According to the American Heritage Dictionary of the English Language, a *cluster* is *a group of the same or similar elements gathered or occurring closely together.*

Comprehension

According to the American Heritage Dictionary of the English Language, comprehension is *the act or fact of grasping the meaning, nature or importance of something.*

1. *Program Comprehension.* It is this the process of acquiring knowledge about a software system.
2. *Specification Comprehension.* Specification comprehension is the process of comprehending specifications - potentially using visualization techniques.

Cliché

1. *Program Cliché.* According to [BF99] a *program cliché* is a *basic knowledge unit used by programmers to build and recognize code.* Clichés are commonly-used computational structures. Examples are data-structure clichés such as stacks, queues and hash tables and algorithmic clichés such as sorting or binary search. Clichés can include fragments of code and further clichés.
2. *Specification Cliché.* A *specification cliché* is a basic knowledge unit used to build and recognize specification text. Specification clichés can consist of both fragments of specification code and intermediate specification clichés (from which further specification clichés may be inferred).

Fragment

According to the American Heritage Dictionary of the English Language it is a *small part broken off or detached.*

1. A *program fragment* is an incomplete or isolated portion of code.
2. *Specification Fragment* A specification fragment is an incomplete or isolated portion of specification code. It 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.

Mapping

In mathematics a mapping is a rule of correspondence established between sets that associates each element of a set with an element in the same or another set.

Partial Specification

A *partial specification* is specification fragment, representing a state space and a set of operations.

Partition

A *partition* is a part or section into which something (a program, a specification, ...) has been divided - it is the decomposition of a set into a family of disjoint sets.

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.

In specification languages these prime objects can be expressions or predicates, but they can also be generic type or schema type definitions. Prime objects are not only restricted to simple expressions. As they form logical units, prime objects can be combined together in order to form so-called *higher-level primes*.

Program Plan

Plans (sometimes called schematas) describe relevant actions or goals of pieces of programs. According to [BRS+97], *plans* are closely related to chunks, *representing stereotypical programming sequences within a framework of task-related goals*. According to [You96b], there is no real difference between the term plan and cliché, as both of them are representing generic knowledge structures that guide the comprehender's interpretation.

Slice

According to the Webster's Encyclopedic Unabridged Dictionary, a slice is *a thin, broad, flat piece cut from something ... a part; portion.*

1. *Program Slice.* The original concept goes back to the PhD thesis from M. Weiser 1979 [Wei79]: He defined a program slice S as a *reduced, executable program obtained from a program P by removing statements, such that S replicates parts of the behavior of P*. Nowadays the definition seams to be weakened a little bit. Nevertheless, in this work the original definition is used, which means, that a slice always has to be (in the context of a programming or specification language) syntactically correct.
2. *Specification Slice.* (a) According to [CR94], a *specification slice* is a specification fragment, that is syntactically and semantically correct. (b) In this work the definition of a specification slice is refined: 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 point of interest.

Specification Text

Specification text (sometimes also called specification code) denotes the literals the specification is written in. For Z or VDM, specifications are usually typeset by using LaTeX.

Transformation

A transformation is a mapping of one space onto another or onto itself. There is a lot of different types of transformations (Laplace transformation, viewing transformation, projective transformation, denotational transformation, ...). If not otherwise stated the term transformation signifies the change of representation of (parts of) specifications.

Specification View

A *specification view* is defining states and operations. According to [Jac95a], a specification view is a *... partial specification of a program. A full specification is then obtained by composing several views, linking them through their states ... and their operations.* The important aspect is, that views can be composed more freely (compared to modules), and operation may appear in more than one view.

Visualization

The term *visualization* is often misunderstood in literature as it contains the root word "visual" (Latin for "sight"). But visualization is not necessarily related to the visual fields. According to the Webster's Encyclopedic Unabridged Dictionary (as well as the Oxford English Dictionary) visualization suggests the *formation of a mental image.* If not otherwise stated, this definition is used in this work.

1. *Software Visualization.* According to [SDB+98], software visualization is the *use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software.*
2. *Specification Visualization.* This term denotes the *visualization of specification code and/or specification dependencies in either static or dynamic form.*

BIBLIOGRAPHY

[ABL96] J.R. Abrial, E. Borger, and H. Langmaack, editors. *Formal Methods for Industrial Applications. Lecture Notes in Computer Science*, volume 1165. Springer Prublications, 1996.

[Alb79] A. J. Albrecht. Measuring application development productivity. In *Proceedings of the IBM Joint SHARE/GUIDE Symposium*, pages 83–92, 1979.

[AP98] V.S. Alagar and K. Periyasamy. *Specification of Software Systems.* Springer, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffry D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison - Wesley, 1986.

[Bal01] Helmut Balzert. *Lehrbuch der Software-Technik. Software-Management, Software Qualitätssicherung, Unternehmensmodellierung.* Spektrum Akademischer Verlag, 2001.

[Bas80] Victor R. Basili. Quantitative software complexity models: A panel summary. In Victor R. Basili, editor, *Tutorial on Models and Methods for Software Management and Engineering.* IEEE Computer Society Press, Los Alamitos, CA, 1980.

[BB95] 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.

[BC85] J.-F. Bergeretti and B. Carré. Information-flow and data-flow analysis of while programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, 1985.

[BDS98] 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. Institute for Operations Research and the Management Sciences, April 1998.

[Bei95] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software Systems*. John Wiley & Sons, Inc., 1995.

[BF99] 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.

[BG94] Jonathan P. Bowen and Michael J. C. Gordon. Z and hol. In J. P. Bowen and J. A. Hall, editors, *8th Z Users Workshop (ZUM), Cambridge 1994*, Workshops in Computing, pages 141–167. Springer-Verlag, 1994.

[BH95] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

[BH97] Jonathan P. Bowen and Michael G. Hinchey. Formal models and the specification process. In JR. Allen B. Tucker, editor, *Computer Science and Engineering Handbook*, pages 2302–2322. CRC Press, 1997.

[BH98] Marc H. Brown and John Hershberger. *Software Visualization – Programming as a Multimedia Experience*, chapter Software Auralization. MIT Press, 1998.

[BKP98a] B. Balzer, T. Katayama, and D. Perry, editors. *Proc. International Workshop on Principles of Software Evolution (IW-PSE98)*, 1998.

[BKP98b] B. Balzer, T. Katayama, and D. Perry, editors. *Proc. International Workshop on Principles of Software Evolution (IW-PSE98)*, 1998.

[BM03] 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.

[Boe80] Barry Boehm. Developing small scale application software projects: Some experimental results. In B. Boehm, editor, *Proceedings, IFIP 8th World Computer Congress*, 1980.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs NJ, USA, 1981.

[Bol02] Andreas Bollin. Specification transformation as a basis for specification comprehension. In *Proceedings of Applied Informatics 02*. AACE, 2002.

[Boo96] Gramercy Books, editor. *Webster's Encyclopedic Unabridged Dictionary*. Random House, 1996.

[Bow96] Jonathan Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press (ITCP), 1996.

[Bow00] Jonathan Bowen. The Z Notation. http:// archive.comlab.ox.ac.uk/ z.html, December 2000.

[Bro78] Ruven Brooks. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd international conference on Software engineering*, pages 196–201, 1978.

[BRS+97] 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.

[BS00] Barry W. Boehm and Kevin J. Sullivan. Software economics: A roadmap. In *The Future of Software Engineering, 22nd International Conference on Software Engineering*, pages 319–344, 2000.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC99] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering: An International Journal. Kluwer Academic Publishers*, 6(1):69–95, January 1999.

[CDHW93] 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.

[CNS91] 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.

[Coo82] M.L. Cook. Software metrics: An introduction and annotated bibliography. *ACM SIGSOFT Software Engineering Notes*, 8:41–60, 1982.

[CR94] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[CSM+79] B. Curtis, S.B. Sheppard, P. Milliman, M.A. Borst, and T. Love. Measuring the psycological complexity of software maintenance tasks with the halstead and mccabe metrics. *IEEE Transactions on Software Engineering*, 5:96–104, 1979.

[CW96] 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.

[dA00] Luca del Alfaro. Mocha homepage. http:// www-cad.eecs.berkeley.edu/ ∼mocha/, December 2000.

[Dav97] Alan M. Davis. *Software Engineering Project Management*, chapter Software Life Cycle Models, pages 105–113. IEEE Computer Society, 1997.

[DDZ96] J. Dautermann, E. Dubinsky, and R. Zazkis. Coordinating visual and analytical strategies: A study of students' understanding iof the group d4. *Journal for Research in Mathematics Education*, 27(4):435–457, 1996.

[Dil99] Antoni Diller. *Z - An Introduction to Formal Methods*. John Wiley and Sons, 1999.

[Edm99] Bruce Edmonds. *Syntactic Measures of Complexity*. PhD thesis, University of Manchester, 1999.

[FF96] Kate Finney and Norman Fenton. Evaluating the effectiveness of z: The claims made about cics and where we go from here. *Journal of Systems and Software*, 35(3):209–216, 1996.

[FK89] 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, 1989.

[FL96] Brigitte Froehlich and Peter Gorm Larsen. Combining vdm-sl specifications with c++ code. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Co-Sponsored by IFIP WG 14.3, Oxford, UK, March 18-22, 1996, Proceedings*, volume 1051 of *Lecture Notes in Computer Science*, pages 179–194, 1996.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[FP97] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics. A Rigorouse & Practical Approach.* PWS Publishing Company, 1997.

[FPB95] Jr. Frederick Philips Brooks. *The mythical man-month: essays on software engineering - Anniversary edition.* Addison Wesley, 1995.

[GH83] John V. Guttag and James J. Horning. An introduction to the larch shared language. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress*, pages 809–814. North-Holland/IFIP, September 19–23 1983.

[Gla03] Robert L. Glass. *Facts and Fallacies of Software Engineering.* Addison-Wesley, 2003.

[GMS94] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion.* Addison Wesley Professional, 1994.

[GWM+93] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ.* Cambridge, 1993.

[GZ99] Narasimhaiah Gorla and Kang Zhang. Deriving program physical structures using bond energy algorithm. In *Proceedings of the Sixth Asia Pacific Software Engineering Conference.* IEEE, December 1999.

[Hal77] M.H. Halstead. *Elements of Software Science*. Elsevier North-Holland, New York, 1977.

[Hal90] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[Hal96] J. Anthony Hall. Using formal methods to develop an atc information system. *IEEE Software*, pages 66–76, March 1996.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HCK+95] D. Hamilton, R. Covington, J. Kelly, C. Kirkwood, M. Thomas, A.R. Flora-Holmquist, M.G. Staskauskas, S.P. Miller, M. Srivas, G. Cleland, and D. MacKenzie. Experiences in applying formal methods to the analysis of software and system requirements. In *1st Workshop on Industrial-Strength Formal Specification Techniques*. IEEE, 1995.

[HDR+96] 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.

[HK81] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

[Hoa85] C.A.R Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[HST97] Daniel Hazel, Paul Strooper, and Owen Traynor. Possum: An Animator for the SUM Specification Language. Technical Report 97-10, University of Queensland, February 1997.

[HST98] Daniel Hazel, Paul Strooper, and Owen Traynor. Requirements Engineering and Verification using Specification Animation. Technical Report 99-26, University of Queensland, June 1998.

[Hum89] Watts Humphrey. *Managing the Software Process*. Addison-Wesley, Reading, Mass., 1989.

[IEE91] IEEE Software Engineering Standards Collection. In IEEE standard glossary of software engineering terminology. Elsevier Applied Science, 1991.

[Ins90] Institute of Electrical and Electronics Engineers, New York, NY. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, 1990.

[Jac95a] Daniel Jackson. Structuring Z Specifications with Views. *ACM Trans. on Software Engineering and Methodology*, 4(4), October 1995.

[Jac95b] Michael Jackson. The world and the machine. In *Proc. 17th International Conference on Software Engineering*, pages 283–292. IEEE-CS Press, 1995.

[Jac96] Jonathan Jacky. *The way of Z.* Cambridge University Press, 1996.

[JDM01] Lamia Labed Jilani, Jules Desharnais, and Ali Mili. Defining and applying measures of distance between specifications. *IEEE Transactions on Software Engineering (TSE)*, 27(8):673–703, August 2001.

[Jen97] Kurt Jensen. *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use - Volume 1.* Springer Verlag, 2nd edition, 1997.

[Jon78] T. C. Jones. Measuring programming quality and productivity. *IBM Systems Journal*, 17(1):39–63, 1978.

[Jon86] C. Jones. *Programming Productivity.* McGraw–Hill, 1986.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, second edition, 1990.

[Jun02] Stefan Jungmayr. Identifying test-critical dependencies. In *Proceedings of the International Conference on Software Maintenance (ICSM'02).* IEEE Press, 2002.

[Kad02] Gert Kadunz. Visualisierung. die verwendung von bildern beim lernen von mathematik. habilitationsschrift. (in german). Abteilung fr Didaktik der Mathematik, Institut fr Mathematik, Universitt Klagenfurt, Klagenfurt, 2002.

[KM99] Claire Knight and Malcolm Munro. Visualising software - a key research area. In *International Conference on Software Maintenance 1999 (ICSM'99)*, August 1999.

[KMW98] Olaf Kummer, Daniel Moldt, and Frank Wienberg. A framework for interacting design/cpn- and java-processes. Technical report, Universit"at Hamburg, Fachbereich Informatik, 1998.

[KPB98] Peter Kokol, Vili Podgorelec, and Janez Brest. A wishful complexity metric. In *Proceedings of The European Software Measurement Conference FESMA98 - Business Improvement through Software Measurement*, pages 235–242, 1998.

[KPHR99] Peter Kokol, Vili Podgorelec, Henri Habrias, and Nassim Hadj Rabia. The complexity of formal specifications - assessments by alpha - metric. *ACM SIGPLAN Notices*, 6:84–88, 1999.

[KST+85] Joseph K. Kearney, Robert L. Sedlmeyer, William B. Thompson, Michael A. Adler, and Michael A. Gray. Problems with software complexity measurement. In *Proceedings of the 1985 ACM Computer Science Conference*, pages 340–347, March 1985.

[KSW96] 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.

[Lak97] Arun Lakhotia. Rule-based approach to computing module cohesion. In *Proceedings of the 15th International Conference on Software Engineering*, pages 35–44. IEEE Computer Society Press, 1997.

[Lev91] Nancy G. Leveson. Software safety in embedded systems. *Communications of the ACM*, 34(2):35–46, 1991.

[LJK+01] Jong Kook Lee, Seung Jae Jung, Soo Dong Kim, Woo Hyun Jang, and Dong Han Ham. Component identification method with coupling and cohesion. In *Proceedings of the Eight Asia-Pacific Software Engineering Conference (APSEC'01)*, 2001.

[LL91] Peter Gorm Larsen and P.B. Lassen. An executable subset of meta-iv with loose specification. in vdm'91: Formal software development methods. *Lecture Notes in Computer Science*, 551, 1991.

[LvH85] D. C. Luckham and F. von Henke. An overview of anna, a specification language for ada. *IEEE Software*, 2:9–23, March 1985.

[LW98] Bo Lindstrom and Lisa Wells. Simulation based performance analysis in design/cpn. Technical report, Department of Computer Science, University of Aarhus, DK-8000 Aarhus C, Denmark, 1998.

[MB03] Roland T. Mittermeir and Andreas Bollin. Demand-driven specification partitioning. In *Proceedings of the 5th Joint Modular Languages Conference, JMLC'03*, Ausgust 2003.

[MBPRR01] 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.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[McC89] Thomas J. McCabe. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1989.

[Mil88] Everald E. Mills. *Software Metrics, SEI Curriculum Module SEI-CM-12-1.1.* Carnegie Mellon University, December 1988.

[Mil89] Robin Milner. *Communicating and Concurrency.* International Series in Computer Science. Prentice-Hall International, London, 1989.

[Mil98] Everald E. Mills. Metrics in the software engineering curriculum. *Annals of Software Engineering 6*, pages 181–200, 1998.

[Mit00] R. Mittermeir. Comprehending by varying focal distance. International Conference on Software Engineeting, ICSE-IWPC2002, June 2000.

[ML87] Michael Marcotty and Henry Ledgard. *The World of Programming Languages.* Springer Books on Professional Computing. Springer Verlag, 1987.

[MRW77] J.A. McCall, P.K. Richards, and G.F. Walters. Factors in software quality. Technical report, Rome Air Development Center, 1977.

[MTO+92] H.A. Müller, S.R. Tilley, M.A. Orgun, B.D. Corrie, and N.H. Madhavji. A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models. In *SIGSOFT'92: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, volume 17 of *5*, pages 88–98. ACM Software Engineering Notes, December 1992.

[Mye77] G.J Myers. An extension to the cyclomatic measure of program complexity. *ACM Sigplan Notices*, 12(10):61–64, 1977.

[Mye97] Brad A. Myers. *The Computer Science and Engineering Handbook*, chapter 72, pages 1571–1595. CRC Press, ACM, 1997.

[NLBN00] 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.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Merkus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer Verlag, 2002.

[OA93] 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.

[OJP99] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic Domain Reduction Procedure for Test Data Generation. *Software – Practice and Experience*, 29(2):167–193, February 1999.

[O'N93] Micheal B. O'Neal. An empirical study of three common software complexity measures. In *Symposium on Applied Computing (SAC-93)*, pages 203–207, 1993.

[OO84] K. Ottenstein and I. Ottenstein. The program dependence graph in a software development environment. In *ACM SIGSOFT/SIGPLAN*, volume 19 of *Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACM, 1984.

[OT89] Linda M. Ott and Jeffrey J. Thus. The relationship between slices and module cohesion. In *11th International Conference on Software Engineering*, pages 198–204, 1989.

[OWE94] M.B O'Neal and Jr. W.R. Edwards. Complexity measures for rule-based programs. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):669–680, 1994.

[Ped00] Jan Storbank Pedersen. Raise homepage. http:// spd-web.terma.com/ Projects/ RAISE/, December 2000.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Technische Hochschule Darmstadt, 1962.

[PH97] Shari Lawrence Pfleeger and Les Hatton. Investigating the influence of formal methods. *IEEE Computer*, 30(2):33–43, Feb. 1997.

[PJCK97] Shari Lawrence Pfleeger, Ross Jeffery, Bill Curtis, and Barbara Kitchenham. Status report on software measurement. *IEEE Software*, 14(2):33–43, March/April 1997.

[PMRR98] H. Pirker, R.T. Mittermeir, and D. Rauner-Reithmayer. Service channels - purpose and tradeoffs. In *Proceedings of the 22nd International Computer Software and Application Conference*, pages 204–211. IEEE, 1998.

[RB87] H.D. Rombach and V.R. Basili. Quantitative software-qualitätssicherung. *Informatik-Spektrum*, 10:145–158, 1987.

[Rei85] W. Reisig. *Petri-nets: An Introduction*. Springer Verlag, 1985.

[Rug95] Spencer Rugaber. Program comprehension. In Marcel Dekker, editor, *Encyclopedia of Computer Science and Technology*, volume 35 of *20*, pages 341–368. Inc:New York, 1995.

[RW90] Charles Rich and Linda M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, January 1990.

[RW02] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *10th International Workshop on Program Comprehension (IWPC'02)*, June 2002.

[SBE83] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, 26(11):853–860, November 1983.

[SC02] 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.

[SDB+98] John Stasko, John Domingue, Mark H. Brown, Blaine A. Price, et al. *Software Visualization Programming as a Multimedia Experience*. MIT Press, 1998.

[SFM99] M.-A.D. Storey, F.D. Fracchia, and H.A. Müller. Cognitive Design Elements to support the Construction of a mental model during Software Visualization. *Journal of Software Systems, special issue on Program Comprehension*, 44:171–185, 1999.

[She93] Martin Shepperd. *Software Engineering Metrics*, volume 1. McGraw-Hill, 1993.

[SM96] Margaret-Anne D. Storey and H.A. Müller. *Software Visualization*, chapter Manipulating And Documenting Software Structures, pages 244–263. World Scientific Publishing Co., 1996.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[SND87] 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.

[Spi89a] J.M. Spivey. An introduction to z and formal specifications. *Software Engineering Journal*, 4(1):40–50, January 1989.

[Spi89b] J.M. Spivey. *The Z Notation*. C.A.R. Hoare Series. Prentice Hall, 1989.

[SW93] J. Stasko and J Wehrli. Three-dimensional computation visualization. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 258–264, 1993.

[SWF+96] M.-A.D. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H.A. Müller. On Designing an Experiment to Evaluate a Reverse Engineering Tool. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE96)*, pages 31–40, November 1996.

[SWH98] M.-A.D. Storey, K. Wong, and H.A.Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? Technical report, School of Computing Science, Simon Fraser University, 1998.

[Tai84] 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.

[Til95] Scott R. Tilley. Domain-retargetable reverse engineering iii: Layered modeling. Technical report, Software Engineering Institute, Carnegie Mellon University, 1995.

[Tip94] Frank Tip. A Survey of Program Slicing Techniques. Technical report, CWI Netherlands, 1994.

[TMO92] Scott R. Tilley, Hausi A. Müller, and Mehmet A. Orgun. Documenting software systems with views. In *SIGDOC'92: Proceedings of the 10th International Conference on Systems Documentation*, pages 211–219. ACM, October 1992.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. 22nd Internat. Conference on Software Engineering*, pages 107 – 119. ACM and IEEE press, May 1999.

[Tuc96] Allen B. Tucker. *The Computer Science and Engineering Handbook*. CRC Press, 1996.

[Tur86] David A. Turner. An overview of miranda. *ACM SIGPLAN Notices*, 21:158–166, 1986.

[Uht91] Augustus K. Uht. A theory of reduced and minimal procedural dependencies. *IEEE Transactions on Computers*, 40(6):681–692, June 1991.

[URN02] 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.

[Utt94] Mark Utting. Animating Z: Interactivity, Transparency and Equivalence. Technical report, University of Queensland, 1994.

[Utt00] Mark Utting. Formal Methods Links. http:// www.cs.waikato.ac.nz/ ∼marku/ formalmethods.html, December 2000.

[VLK98] 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.

[vMV94] A. von Mayrhauser and A. M. Vans. Program Understanding - A Survey. Technical Report CS-94-120, Colorado State University, 1994.

[vV93] Hans van Vliet. *Software Engineering Principles and Practice*. John Wiley & Sons, 1993.

[Wat01] Geoffry Norman Watson. *A Generic Proof Checker*. PhD thesis, The School of Computer Science. The University of Queensland, 2001.

[WC03] Laurie Williams and Alistair Cockburn. Agile software development: It's about feedback and change. *IEEE Computer*, pages 39–43, June 2003.

[WD96] Jim Woodcock and Jim Davis. *Using Z - Specification, Refinement, and Proof*. C.A.R. Hoare Series. Prentice Hall International, 1996.

[Wei79] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.

[WHH79] M. R. Woodward, M. A. Henell, and D. Hedley. A measure of control flow complexity in program text. *IEEE Transaction on Software Engineering*, 5(1):45–50, 1979.

[Wig97] 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.

[WM87] Michael W.Evans and John Marciniak. *Software Quality Assurance and Management*. John Wiley & Sons, Inc., New York, NY, 1987.

[Won98] Kenny Wong. *Rigi User's Manual*. Department of Computer Science University of Victoria, June 1998.

[YM98] Peter Young and Malcom Munro. Visual software in virtual reality. In *International Workshop on Program Comprehension 1998*. IEEE, 1998.

[You96a] Peter Young. Program comprehension. Visualisation Research Group. http:// vrg.dur.ac.uk/ misc/ PeterYoung/ pages/ work/ documents/ lit-survey/ prog-comp, May 1996.

[You96b] Peter Young. Software Visualization. Technical report, Centre for Software Maintenance, University of Durham, June 1996.

[ZCU96] 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.

[ZCU97] Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima. Slicing concurrent logic programs. In T. Ida, A. Ohori, and M. Takeichi, editors, *Second Fuji International Workshop on Functional and Logic Programming*, pages 143–162. World Scientific, 1997.

[Zha96] Jianjun Zhao. *Program Dependence Analysis of Concurrent Logic Programs and Its Application*. PhD thesis, Kyushu University, December 1996.

[Zha99] Jianjun Zhao. Extracting reusable software architectures: A slicing-based approach. In *ESEC/FSE'99 Workshop on Object-Oriented Reengineering Toulouse (France)*, September 1999.

INDEX