The Efficiency of Specification Fragments

Andreas Bollin
Institute for Informatics-Systems
University of Klagenfurt, Austria
Andreas.Bollin@uni-klu.ac.at

Abstract

Formal specifications are valid sources for comprehension tasks when used during later development phases. However, the linguistic density of specification languages and the size of specifications can still be seen as an obstacle against comprehension activities.

This paper presents an approach for the identification of fragments of Z specifications with a well defined semantic content. These fragments, specification chunks and specification slices, are analyzed in respect to their efficiency when used during typical comprehension tasks.

1. Introduction

Formal methods and the application of formal specification languages play a crucial role in software engineering. During initial phases they are recommended as means to produce high quality software. However, using formal specifications at early steps exclusively is unfavorable in so far, as they can also be used as important drivers for test data generation [1]. When specifications are kept up to date during the various evolutionary steps of system development they can also play a vital role during reverse engineering activities and the identification of concepts [13].

It is not without effort to keep specifications constantly up to date. The effort only pays off when additional benefits can be obtained. Such a benefit can be achieved by speeding up specification comprehension.

It has already been argued in [11] that the density of expressing thoughts is a positive attribute during development (from the perspective of the specification's writer). But this density becomes detrimental for specification comprehension during later phases. This is even worse when specifications are getting really large and the bulk of information exceeds dozens (if not hundreds) of pages of specification code. It is this observation of compactness that confirms the myth that formal specifications do not scale up.

The inherent density of specification languages cannot be changed. But comprehension problems aggregate when problem-inherent complexity is combined with the complexity of size. The latter can be reduced if a formal mechanism is devised which ensures that the user of a specification is presented only the portion of the specification (in the latter called specification fragment) relevant to the particular problem at hand [2].

In code comprehension, slices [17] and chunks [6] have been proposed as mechanisms for the identification of well defined portions of code. Both approaches guarantee that all that needs to be studied for the problem at hand is presented to the user. The notion of a specification slice has been introduced in [12], however, several types of specification fragments (to which specification slices and chunks are belonging to) are formally defined in [3] for the first time.

It is a legitimate question whether the approach of generating specification fragments can play the same role during reverse engineering steps as slices and chunks do. However, the usefulness depends on the effects (and limits) gained by the approach, and it has to be clear what can be expected when generating specification fragments.

As a first step this work focuses on the complexity of specifications – and with it on the complexity of the underlying comprehension task. The evaluation presented in the second part of this paper demonstrates that, especially with larger specifications, the generation of specifications fragments is very efficient. Complexity and size is reduced in a practicable manner, which is a strong evidence for the usefulness of the approach.

The paper is organized as follows: the subsequent section introduces different types of specification fragments and discusses the detection of dependencies necessary for the generation of slices and chunks. Then the complexity of specifications is discussed and a basis for the calculation of complexity measures is provided. The paper then examines the efficiency of the approach of generating specification fragments, and it closes with a discussion of the results of the experimental studies.

$[\mathit{NAME}, \mathit{DATE}]$

BB

$\mathit{known} : \mathbb{P}\,\mathit{NAME}$
$\mathit{birthday} : \mathit{NAME} \pfun \mathit{DATE}$

$\mathit{known} = \operatorname{dom} \mathit{birthday}$

Add

$\Delta \mathit{BB}$
$\mathit{name}? : \mathit{NAME}$
$\mathit{date}? : \mathit{DATE}$

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

Figure 1. Fragment of the Z Birthday-Book specification out of [7]. It specifies a system for storing names (*NAME*) and birthday dates (*DATE*) in a database called *BB*. The *Add* operation schema takes a name and a date as arguments and stores them in the database.

2. Specification Fragments

The proper support of specification comprehension activities implies to provide developers and maintenance personnel with partial specifications that

- are substantially smaller than the full specification,
- contain all relevant parts of interest, and that
- can be automatically derived from the original specification.

The informatics literature is full of concepts of partiality, concepts that aim to provide an interested party just the perspective needed for a particular task. The notions of views [8], slices [16, 17], and chunks [6] come to mind, and it seems obvious to map those concepts to specifications.

Specifications and programs are different and mapping decomposition concepts from programming languages to declarative languages is a challenging part. There is an explicit flow of control with imperative programs. Line-numbers represent an explicit order among statements or other constructs and it is state of the practice to apply techniques like constructing a PDG for further dependency analysis.

On the other hand, when looking at declarative specification languages like VDM [9] or Z [14], there is no explicit flow of control. The notion of control dependency has to be re-interpreted. Thus, when talking about the decomposition of a specification, one has to be careful in putting specifications on a par with imperative programs.

As the assessment of the approach is based on the notions of specification slices and chunks, these concepts are presented in the remainder of this section. Readers interested in a more formal set of definitions are referred to [4, 2].

2.1. Elements of Formal Specifications

Formal specifications are expressions written in some formal language. Each specification language consists of linguistic elements, and the primitives of the language have a formally defined semantics. These primitives are referred to as *literals* of the language. Examples of literals are identifiers or linguistic operators, such as the literal "$\mathbb{P}$", the identifier "*NAME*" or the operator "dom" in the birthday book specification that is represented in Fig. 1.

Minimal and meaningful linguistic expressions are called *prime objects* or *primes* for short. They are syntactic elements of specifications with a formally defined semantics. Such primes are constructed from literals and form the basic entities of specifications. Examples are declarations like "$\mathit{name}? : \mathit{NAME}$" or expressions such as "$\mathit{name}? \notin \mathit{known}$" in the *Add* operation schema of the specification in Fig. 1.

One important aspect is that primes are immutable as far as they constitute fundamental units (e.g. states and operations) that specifications are built upon. Several specification languages permit semantically richer primes (the *Add* operation schema is such a fundamental unit). These primes are called *higher level primes* and are well-defined arrangements of literals and other primes.

It is not sufficient to support the comprehension process by just looking for independent prime objects. During comprehension tasks the user of a specification needs more than a given prime but less than the full specification. As the needed information is distributed on the two-dimensional, linear text, one needs a set of different primes which are to be found in different parts of the specification. Such a set, consisting of different but related primes is called *specification fragment*. Such a fragment might be the unboxed Z paragraph "$[\mathit{NAME}, \mathit{DATE}]$" and the set consisting of the two primes "$\mathit{known} : \mathbb{P}\,\mathit{NAME}$" and "$\mathit{birthday} : \mathit{NAME} \pfun \mathit{DATE}$". This fragment of the specification in Fig. 1 grasps the state space of the specification without telling anything about the relationship between *known* and *birthday*.

With these (informal) definitions, specification languages can be structured into elements having defined semantics on their own (primes) and elements which obtain their semantics from their arrangement in a broader context [11].

2.2. Specification Slices and Chunks

The previous section focused on syntactic components of specifications. In the scope of comprehension activities some higher-level abstraction with *clearly defined* semantics are needed and it makes sense to start constructing these abstractions from well-known specification elements, namely from specification primes. Such higher-level (or semantic-based) specification concepts are, among others, the already mentioned concepts of slices and chunks.

Both, slices and chunks, are abstractions of the given specification defined ”around” a specific point of interest. In analogy to the slicing criterion of program slicing, this point of interest is called *abstraction criterion*. It denotes a specific prime or a set of primes of the specification.

The basic mechanism for the identification of slices and chunks is, as with most program comprehension approaches, based on the identification of control (and data) dependencies. However, state-based specifications are in general different from imperative programs. Control is not predominant, there are no line numbers and the ordering of predicates is irrelevant, thus primes are not “lined up” by flow of control.

When looking closer at specifications it can be observed, though, that parts of a specification are in essence controlled by other parts of it: post-conditions are dependent on pre-conditions. In languages where pre-conditions are explicitly highlighted, this is evident. In other languages, such as *Z*, one may resort to theorem proving techniques to identify pre-conditions.

The calculation of the relevant pre- and post-conditions is a time-consuming task, but the calculation can be skipped by taking before-state predicates as pre-conditions and after-state predicates as post-conditions directly. Before-state and after-state identifiers can be identified on a syntactical basis efficiently. For Z it is shown in [4] that this syntactic approximation is quite accurate. In Fig. 1 the identifier “$birthday'$” in the *Add* operation schema denotes an after state (it is a primed identifier), and thus the predicate “$birthday' = birthday \cup \{name? \mapsto date?\}$” is a post-condition prime. As there is no after-state identifier in the predicate “$name \notin birthday$”, this prime is said to be a pre-condition prime[1].

Control dependencies in Z are located in a schema between pre- and post-condition primes, but control dependencies are also to be found in schemata which are combined via schema operations. Here, again some approximation is conducted. The semantic analysis is skipped, and pre- as well as post-condition expressions are reduced to sets of before- and after-state primes.

By following these simplifications, definitions for control, data, and syntactical dependencies can be provided:

Definition 1 *A specification prime p is* **syntactical dependent** *on a specification prime q, if q is needed to keep p syntactically correct.*

Definition 2 *A specification prime p is* **control dependent** *on a specification prime q, if q potentially decides whether p applies or not.*

Definition 3 *A specification prime p is* **data dependent** *on a specification prime q, if data potentially propagates from q to p through a series of state changes.*

In the specification of Fig. 1 the prime “$birthday' = \ldots$” of the *Add* operation schema is control dependent on the prime “$name? \notin known$” as this prime decides whether the state is changed or not. More complex examples are presented in [4].

With the above definition of dependencies within specifications, a mechanism for carving out slices and chunks from formally correct specifications can be provided.

Definition 4 *A* **specification chunk (SChunk())** *is (i) a prime including all primes contained within it or, (ii) a set of primes that exists within the same specification scope. For each pair of primes within this set of primes either one prime is dependent on the other or both primes are dependent on a third prime (within the set of primes).*

The really important thing is that a chunk always has to be comprehensible in isolation. Compared to a specification fragment, a chunk contains enough (surrounding) context to stay understandable. That means that at least enough semantic information has to be taken into consideration when generating specification chunks. To denote those types of dependencies that are to be considered when calculating the specification chunk, the operation name is augmented by a set of characters representing the included dependencies (S..Syntactic-, C..Control-, D..Data-dependencies). The operation “$SChunk_{[SD]}()$” represents a static chunk with respect to syntactic (S) and data (D) dependencies.

Definition 5 *A* **full static specification slice (FSSlice())** *is a syntactically and semantically correct specification which is the result of adding those primes to an (initially empty) specification which are directly or indirectly contributing to the abstraction criterion.*

In contrast to the definitions provided in [7], the approach constructs specification fragments in a bottom up manner. This assures that every abstraction derived by the approach has a well defined semantics. Starting with a

1 Formally it can be shown that this predicate really is part of the post-condition of the *Add* operation schema.

prime (or a set of primes) as abstraction criterion, the abstraction has this prime's (or these primes') semantics. By aggregating further primes properly, the remaining specification fragment always has a well defined semantics.

As with program slices, it is required that a specification slice is a syntactically correct specification and that it is semantically complete. This is obtained, if *all* kinds of dependencies are included in the specification criterion, which means that a full static specification slice can be calculated by calculating the static specification chunk "$SChunk_{[SCD]}()$".

2.3. Augmented Specification Relationship Net

Specification languages have built in clues to convey semantics. As with programming languages, syntax *and* layout of specification languages support the process of creation, but these characteristics are detrimental for detecting programming-language-like dependency types. However, hidden structural properties of written specifications can be detected in analogy to state-of-the-art techniques of other disciplines:

- In differential calculus it is sometimes easier to *transform* an equation into another space, solve the equation there and perform a backward transformation.
- When dealing with programming languages, a program is *transformed* to an abstract syntax tree which enables the construction of a program dependency graph, and which ultimately facilitates program dependency analysis.
- In [18] it is shown that a transformation of concurrent logic programs to a graph is useful for dependency detection. Zhao et. al. also show that such a graph forms a suitable basis for metrics calculation.

Following these ideas, it seems appropriate to transform the specification into a graph in order to analyze and identify dependencies. What is needed is a structure that, on the one hand, fully replaces the original specification, and, on the other hand, eases the identification of dependencies. Additionally the structure should be isomorphic. This guarantees that a transformation and backward transformation is possible in any case, and that it is up to the user which representation s/he chooses for the problem at hand.

A suitable form of representation is that of an augmented net. It can be used to cope with syntactic AND semantic information.

Structural information is captured in a net called *Specification Relationship Net* (*SRN*). Vertices in the SRN represent primes of the specification, and arcs represent relationships among them. Fig. 2 demonstrates how a Z schema

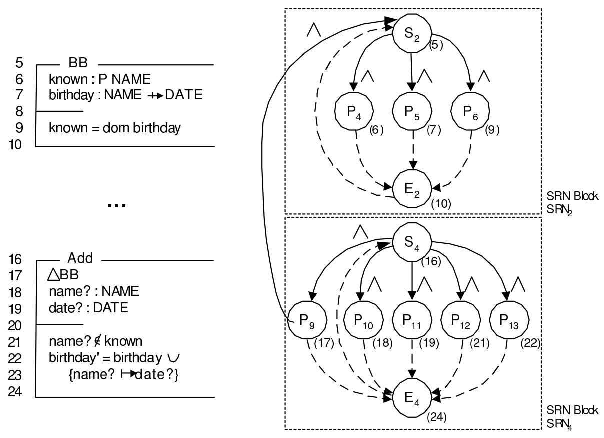


Figure 2. Z specification and SRN representing the birthday book state schema and the *Add* operation schema. Vertices are annotated by line numbers of the specification source. E.g. vertex P_4 represents the predicate prime "*known* : $\mathbf{P}$ *NAME*" at line 6.

is transformed to the SRN representation. Every prime is mapped to a prime vertex, and every schema is enclosed between two special vertices: a start and an end vertex[2]. References to other schemata are expressed by vertices connecting the referring prime to the referred SRN block.

As specifications contain language- and layout-related information, the SRN is extended by vertices representing structural information and comments. This extension depends on the specification language at hand. The same holds for prime objects. The SRN is extended by this information and makes up the *extended SRN (eSRN)*. The transformation to the eSRN has two advantages:

- The eSRN can be used to deal with any information that exists in a specification. The SRN handles every prime and represents the "loose" structure of a specification. The eSRN is able to handle layout information and comments.
- The transformation function itself can be defined in a *bijective* manner. This means that a backward transformation is possible and the two forms of representation can be used interchangeable.

The last step is the augmentation of the net. To ease the identification of dependencies, the eSRN is augmented by

2 As can be seen in Fig. 2 the arcs in the net are classified. There are sequential-, AND- ($\wedge$) and OR- ($\vee$) control arcs used to express different logical and sequential relationships between primes in the specification. See [4] for more details.

declaration, definition and use information of identifiers attached to prime vertices. The ASRN captures the explicit semantics of the specification. Based on reachability conditions, the ASRN is then used to define control, data and syntactical/declarational dependencies in Z specifications. These dependencies then provide the basis for the definition of slices and chunks.

The net itself represents more than just a hidden structure of a specification. It reveals quite a lot of properties of the specification: its size, inter-relationships between primes and number and type of dependencies. It is a candidate for calculating specification metrics and provides the basis for the description of a specification's complexity.

3. The Complexity of Specifications

There are at least two possibilities to assess the effectiveness of the approach of generating and using specification fragments and thus raising their usability for comprehension activities. Firstly, by describing the effects on the complexity of the underlying specification (and therefore on the underlying task) via suitable metrics. Secondly, by conducting empirical studies.

These approaches are not mutually exclusive. However, even in the case of empirical studies, metrics are necessary for the assessment. This section suggests to use the ASRN in order to simplify the calculation of complexity measures and to facilitate the comparison between full specifications and their corresponding specification fragments. The calculation of ASRN related complexity measures is independent of the specification language. However, the approach is applied and validated on the basis of Z specifications.

It is generally agreed that a single measure is not sufficient to represent the overall complexity of a specification. The few existing size-based measures (like lines of specification code or numbers of operators) are not suitable to describe the complexity in its entirety as it is not only size that matters. This argument is easily reinforced as specifications of equal size are not necessarily of the same complexity. It is *logical and structural* complexity that is not to be neglected. The existing set of size-based measures can be extended by looking at the structural information that is explicitly available in the ASRN.

The following complexity measures are easily derived from the augmented specification relationship net:

- Conceptual Complexity $CC(\Psi)$ of a specification Ψ. Here, prime objects of the specification are counted instead of the number of lines of specification code.
- Extended Cyclomatic Complexity $v'(\Psi) = (v(l), v(u))$ of a specification Ψ. $v(u)$ is the upper bound of the tuple and represents the total number of control dependencies in the ASRN. $v(l)$ is the lower bounds and represents the number of pre-condition primes in the specification. This measure can be compared to the extended cyclomatic complexity of programs [10].
- The Definition/Use count metric $DU(\Psi)$ (in analogy to the DU Count metrics presented in [15]) of a specification Ψ. By counting the number of data dependencies the maximum number of data relationships in the ASRN are identified.

With these extensions to the ordinary set of measures detailed comparisons between specifications in respect to several aspects of complexity become feasible for the first time.

4. Specification Comprehension

The approach of calculating specification fragments promises to scale down the complexity of specifications and with it the complexity of the underlying comprehension task. This section provides several small experiments to underpin this statement.

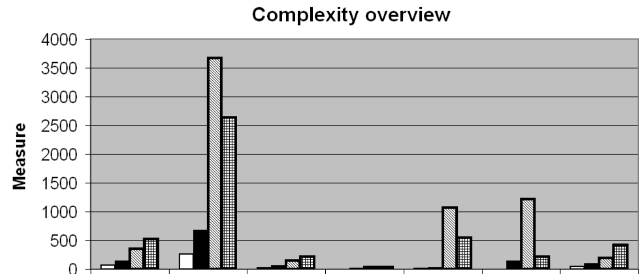
4.1. General Setting

The birthday-book specification is one of the smallest specifications that are to be found in textbooks introducing Z. Larger specifications are necessary in order to express the effects of specification fragments. This includes the influence of the different types of abstraction, the effect of the approach onto complexity in general and the effect of larger specifications.

This section makes use of three additional specifications: The “Elevator” specification out of [7], the “Petrol-Station” specification (*Petrol* for short) which is used during class labs at the University of Klagenfurt and the “ITC Window Manager”-specification (*WM* for short) which is a commercial specification presented in [5].

Fig. 3 summarizes the key attributes of the specifications and visualizes the measures that contribute to their complexity: the total number of vertices (V) in the ASRN, the total number of arcs (A) in the ASRN, the extent of conceptual complexity (CC), the lower and upper bound ($v'(l), v'(u)$) of the extended cyclomatic complexity and the definition/use count (DU) of the specification.

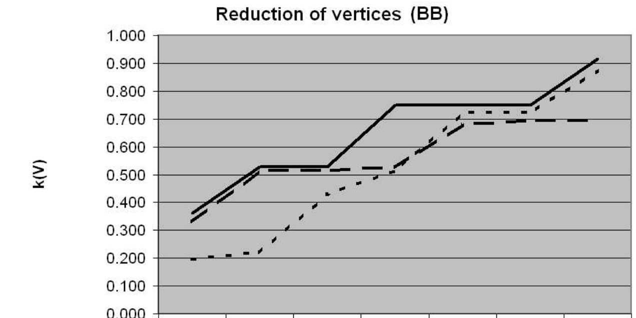
As can be seen in Fig. 3, the *BB*-specification is the most simple one. The *Petrol*-specification is also small, but contains twice as many primes (and about twice as many vertices and arcs). The *BB*-specification consists of 72 vertices, the *Petrol*-specification consists of 134 vertices. The same ratio holds for the other measures, except for the *DU* count



	V	A	CC	v'(l)	v'(u)	DU	LOC
□ BB	72	267	24	4	10	4	48
■ Petrol	134	674	53	10	28	131	87
▨ Elevator	349	3668	144	32	1069	1212	190
▦ WM	520	2633	213	39	544	215	416

Complexity attributes

Figure 3. Complexity overview regarding four Z-specifications. The table summarizes the total number of vertices V and arcs A in the ASRN representation, the conceptual complexity CC, the extended cyclomatic complexity $v' = (v'(l), v'(u))$, the DU (Def/Use) count metric, and the LOC of the specification.



	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

Points of interest

Figure 4. Reduction factor k of vertices after the generation of all 7 specification fragments ($k = V_{fragment}/V_{orig}$) of the BB specification. Generating a slice in respect to prime 1 reduced the total number of vertices (= 72) to 26 – which results in a factor of $k(V)_1 = 0.361$.

metric. Here, the *Petrol*-specification contains 131 data-dependencies, the *BB*-specification contains only 4 data-dependencies.

With respect to the total number of vertices in the ASRN, the *Elevator*-specification is the next larger one. The ASRN contains 349 vertices and is thus smaller than the *WM*-specification. It also contains only 144 primes, whereas the *WM*-specification consists of 213 primes. However, the *Elevator*-specification contains much more hidden dependencies. It contains 1069 control dependencies, whereas the *WM*-specification contains only 544 control dependencies. Thus, with respect to primes, the *Elevator*-specification is less complex than the *WM*-specification. With respect to the extended cyclomatic complexity and even the DU metric, the *WM*-specification is less complex. The same differences can be observed when looking at the number of lines of specification code.

4.2. Experiments

When tracing specifications it depends on the (reverse engineering) problem at hand which primes are relevant to the user and which types of fragments are to be generated. In order to provide a reliable statement about the benefits of the approach an experiment (and no case study) has been designed. The underlying idea is to look at a EVERY prime of the specification and therefor to look at all possible points of interest and all types of specification fragments.

A simple prototype has been implemented and used in order to generate all possible sets of specification fragments for every predicate prime of a given specification. The prototype takes a Z specification (type set in LaTeX) and a slicing/chunking criterion as input. It then generates the specification fragment and (i) exports the fragment into a LaTeX file and (ii) provides feedback about the ASRN representation.

The *BB*-, *Petrol*-, *Elevator* and *WM*-specifications are used as experimental objects for the treatment – the application of the generation of specification fragments. For every point of interest three different types of specification fragments are generated.

In fact, the following steps are conducted during the treatment:

i. For *every* prime vertex representing a point of interest in the specification the full static specification slice $FSSlice()$ is calculated.

ii. For *every* prime vertex representing a point of interest in the specification the full static specification chunk $SChunk_{[S,D]}$ is calculated.

iii. For *every* prime vertex representing a point of interest p in the specification the full static specification chunk $SChunk_{[S,C]}$ is calculated.

For all points of interest slices and two types of chunks (one containing control dependencies, one containing data-dependencies) are calculated. In the *BB*-specification there are 7 points of interest. The *Petrol*-specification contains 21, the *Elevator*-specification contains 102, and the *WM*-specification contains 92 points of interest. Fig. 4 demonstrates the influence of the generation of specification fragments on the number of vertices in the ASRN representing the birthday book specification.

Depending on the specification, this leads to 21 (= 7 points of interest · 3 types of specification fragments) possible fragments for the *BB*-specification, 63 fragments for the *Petrol*-specification, 306 fragments for the *Elevator*-specification and 276 specification fragments for the *WM*-specification.

The factors which are assumed to change (and which are the response variables) are the following:

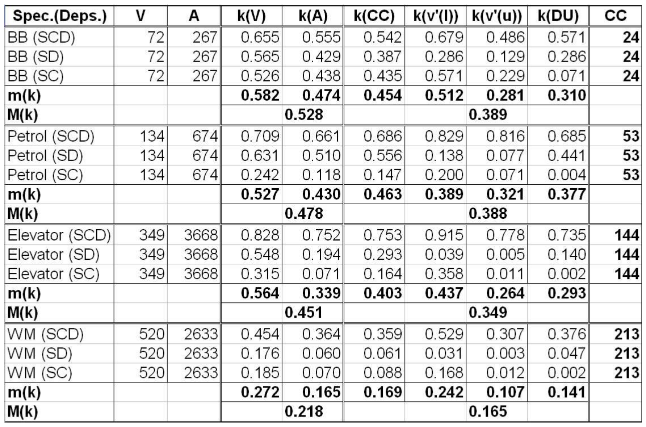
- Attributes regarding the ASRN. They include the total number of vertices *V* and arcs *A* of the net.
- The attribute contributing to the information content of the specification: $CC(\Psi)$, the conceptual complexity.
- Three attributes contributing to the logical complexity of the specification. This includes the lower bound *l* and the upper bound *u* of the extended cyclomatic complexity v' and the *DU* metric of the specification.

These attributes are determined for every specification fragment that is generated during the treatment. For the scope of the evaluation of the approach, the mean values of these attributes (in respect to a specific type of fragment) are considered.

4.3. Specification Fragments' Efficiency

4.3.1. Extent of Reduction of Complexity When generating all types of specification fragments, the first observation is that slices and chunks reduce the size of the specification. As slices include all types of dependencies and as chunks omit dependencies in the resulting specification, it seems to be obvious that, for a specific point of interest, a specification chunk should be smaller than the corresponding specification slice. Fig. 5 visualizes the results of the application of the slicing and chunking approach for all four specifications. It can be observed that there is a reduction of complexity in all cases – which is indicated by values of $k(...)$ that are lower than 1.

Additionally, it can be observed that the values of the reduction factors of chunks are in all cases lower than the values of the reduction factors of the corresponding slices. For the conceptual complexity *CC* of the *BB* specification the extent of reduction is $k(CC) = 0.542$ (when generating slices), whereas the extent of reduction is $k(CC) = 0.387$ (or $k(CC) = 0.435$) when generating chunks. The disadvantage of neglecting information becomes an advantage as the focus gets sharper.



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**		

Figure 5. Table summarizing the mean values *m* of the reduction factors with respect to four specifications and three different types of abstractions. Additionally it provides the mean *M* of the reduction of ASRN and specification attributes. The first column names the specification and the corresponding abstraction criteria. An SCD indicates the generation of a specification slice, SD and SC indicate the generation of a specification chunk.

4.3.2. Influence of the Specification's Size When comparing the extent of the reduction of complexity between the *BB* and the *WM* specification it can be observed that the reduction factors k of the WM specification are generally lower than the reduction factors k of the *BB* specification (which means a higher extent of reduction). Calculating the values for specifications of different sizes leads to a reasonable observation: the effect obtainable by slicing and chunking rises with the size of the specification under consideration.

The assumption is based on the heuristic that slicing and chunking carve out concepts of the specification, concepts that are independent from the specification at hand and are of (roughly) equal size. Thus it is reasonable that with larger specifications the extent of the reduction increases, too.

When talking about the size of a specification, the conceptual complexity (which is equivalent to the number of primes in the specification) is an appropriate measure. However, to describe the average case, the mean values (*M*) of all reduction factors are taken as a basis and Fig. 5 summarizes the dependent and independent variables. It can be observed that the mean values *M*, expressing the average reduction of ASRN and specification attributes, decrease with increasing size of the specifications.

Fig. 6 summarizes the mean M of the mean values of the reduction factors for ASRN metrics and complexity mea-

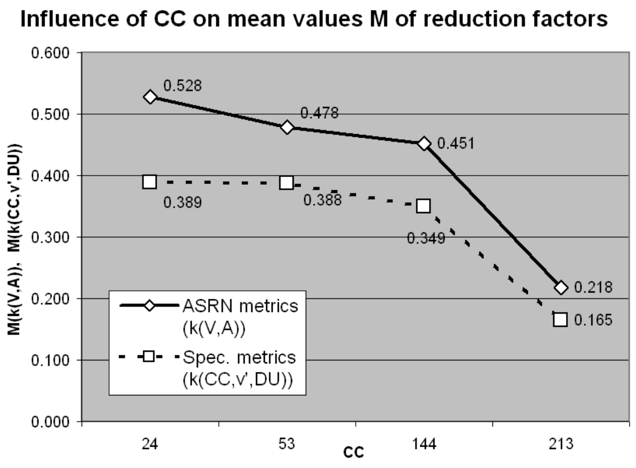


Figure 6. Influence of size on the extent of reduction. The values are based on the results summarized in Fig. 5 and demonstrate that with increasing size of the underlying specification the extent of reduction increases, too.

sures. It can be observed that the larger the specification the higher the overall reduction. The ASRN of the *BB*-specification ($CC = 24$) is reduced by a factor of 1.89 ($= 1/0.528$). The logical complexity is reduced by a factor of 2.57 ($= 1/0.389$). This can be compared with the much larger *WM*-specification ($CC = 213$). Here, the ASRN net is reduced by a factor of 4.59 ($= 1/0.218$). The logical complexity is reduced by a factor of 6.06 ($= 1/0.165$). The mean value of the reduction increases with increasing size of *CC*, thus the approach is more efficient when applied to larger specifications.

Nevertheless, the basic assumption that both approaches carve out concepts from a specification in a similar way does not hold. Slices ensure that all dependencies are considered, chunks allow to neglect existing information. This means that when generating chunks only, the chance is higher to generate smaller specifications. With slices one might have similar problems as with program slicing – the resulting slice has to contain all necessary statements and it is thus as large as the original program.

Generally speaking, the larger the specification, the higher the mean value of the reduction factor of the net and the mean value of the reduction of a specification's complexity. When considering only one type of abstraction, this observation does not hold. Taking a closer look at the generated specification slices, the value of $k(V)$ *increases* with increasing size of the specifications.

4.3.3. Efficiency of the Approach Two aspects are important when generating slices or chunks: the benefits should increase with increasing size of the specifications at hand. This means that the generated specification fragments should stay at a comparable conceptual level. Then, logical complexity should decrease significantly. The effect of scaling down specifications should influence the reduction of hidden dependencies at least with equal size.

For the first aspect not the extents of reduction, but measures contributing to the information content are of interest: the total numbers of vertices (V) and arcs (A) in the ASRN and the conceptual complexity *CC*.

For the second aspect the relationship between conceptual and logical complexity is of interest. Vertices represent conceptual entities and arcs represent logical dependencies. Thus the comparison of conceptual and logical complexity can be carried out by looking at the ratio between the reduction of vertices ($k(V)$) and the reduction of arcs ($k(A)$) in the net. This ratio is described by the factor $f(k)$ which is defined as follows: $f(k) = k(A)/k(V)$. It expresses the ratio between the decrease of vertices and the decrease of arcs in the net. A value lower than 1 indicates that arcs are decreased to a greater extent than vertices. This leads to the following definition:

Definition 6 Efficiency of the approach. *The approach of generating specification fragments is treated* efficient*, iff*
(a.) the mean values of the size of the net and the mean values of the complexity increase at a lower scale than the original specifications.
(b.) the logical complexity (described via v' and DU) is at least reduced to the same extent as the conceptual complexity (described via CC).

The basic assumption is that the reduction of the number of vertices leads to a higher reduction of arcs in the net. Formally, when the ASRN contains n vertices and a arcs, then there are $3 \cdot n^2$ possible arcs (of types C, D and S), at the most, in the net. If decreasing the number of vertices n by a factor of k_v, then the number of arcs should decrease at least by a factor k_a that is equal or higher than k_v. It holds:

$$\begin{aligned} &a = O(3 \cdot n^2) \\ &a' = O(3 \cdot (n/k_v)^2) \qquad\qquad \textit{with } k_v = O(n/(n')) \\ &k_a = a/(a') = O(\tfrac{3 \cdot n^2}{3 \cdot (n/k_v)^2}) = O(k_v^2) \end{aligned}$$

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.

The following observation can be made: The mean value of the sizes of specification fragments (described by a factor *Delta* which is dependent on V', A', CC') increases at a

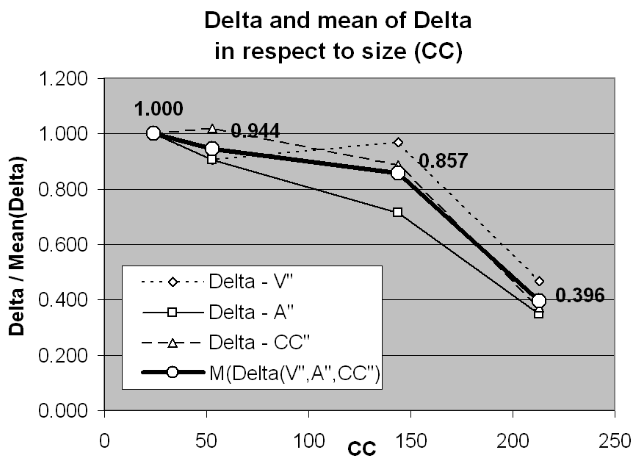


Figure 7. Effect of reduction when increasing the size (CC) of the specification. If the effect increases at the same ratio as the original specification, then the value of $Delta$ should be 1. A value of $Delta$ lower than 1 indicates that the approach is more efficient.

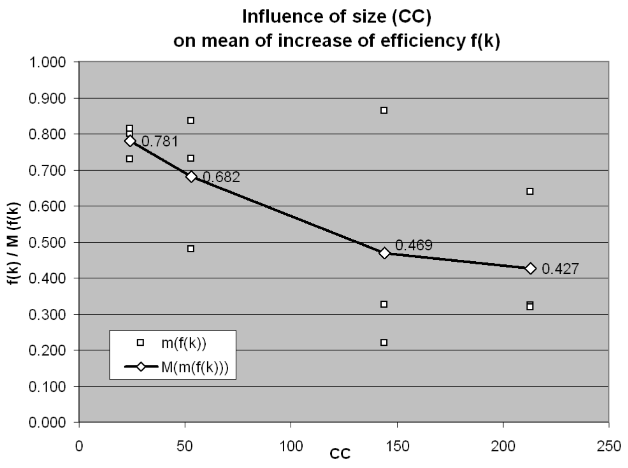


Figure 8. The effect of reduction ($m(f(k))$) increases with the size (CC) of the specification. If the effect increases at the same ratio as the original specification, then the value of $m(f(k))$ should be 1. Additionally, the mean of all reductions ($M(m(fk))$) is presented.

lower scale than the sizes of the underlying specifications. An increase of CC by a factor I results in an increase of size and complexity by a factor less than I. It holds that $M(Delta) \leq 1$. Here, I describes the increase of the specification in respect to vertices, arcs and conceptual complexity. *Delta* expresses the ratio between true values of reduction attributes and estimated values of that attributes. A value lower than 1 indicates that the extent of decrease is higher than the expected extent of decrease.

A value of *Delta* less or equal than 1 indicates that the increase of complexity measures is less or equal than the increase of the size of the specification. This indicates that the approach is efficient in terms of Def. 6. The complexity of the specification fragments increases at lower scale than the size of the underlying specifications. If the value is about 1 then there is still reduction of complexity. However, this implies that, when the size of the specification increases by a factor of 2, the approach generates specification fragments that also increase by a factor of 2.

As can be seen in Fig. 7 there is a decrease in the value of $M(Delta)$ with growing sizes of the specification. Starting with the *BB*-specification the factor is 1.000. The *Petrol*-specification is about twice as large as the *BB*-specification (for the number of vertices $I = 1.8$). However, the fragment contains $V' = 70.667$ vertices in the mean, and not $\tilde{V}' = 77.989$ vertices. The approach has been a bit more effective. Therefore $Delta = 0.906$ which is less than 1. When looking at vertices, arcs and the conceptual complexity, the mean value $M(Delta)$ is 0.944 which is also less than 1. The same holds for the mean values $M(Delta)$ of the *Elevator*- and *WM*-specification.

The second part of the definition of efficiency deals with the ratio between logical and conceptual complexity. It can be observed that the logical complexity is at least reduced to the same extent as the conceptual complexity. Thus when generating all possible fragments it holds that $M(f(k)) \leq 1$.

The factor $f(k)(= k(V)/k(A))$ tells a lot about the ratio between the reduction of vertices and arcs in the net. If $f(k)$ decreases, this indicates that the number of arcs is reduced to a much greater extent than the number of vertices. This underpins the heuristic that, in the average case, the reduction $r_A = 1/k(A)$ increases with at least the order of the reduction of the factor $r_V = 1/k(V)$.

When looking at the factor $f(k)$ for all four specifications it can be observed that all the values are less than 1. This indicates that, in any case, the extent of reduction of arcs is higher than the extent of the reduction of vertices. When looking at the mean of the reductions ($M(f(k))$) it can be observed that, with increasing size of the underlying specification, the extent of the reduction increases, too. The approach gets more and more efficient with larger specifications at hand. Fig. 8 visualizes the values for the factor $f(k)$ (dependent on CC) and the mean $M(f(k))$ of the factors. It also underpins the statement of the positive effects of the approach.

4.3.4. Summary of Observations

Based on the four experimental objects all possible types of specification fragments for all points of interest have been calculated. The following observations have been made:

O1 Specification chunks reduce complexity more than specification slices. This observation holds for all specifications that have been examined so far. Additionally, there is strong evidence that chunks reduce the complexity in more cases than specification slices do.

O2 The effect obtainable by slicing and chunking rises with the size of the specification under consideration. For the mean values of complexity attributes this observation has been confirmed. However, there are a few cases when this observation does not hold. Again, slices do not always lead to smaller specifications, while chunks usually do.

O3 The effect observable by slicing and chunking is significant. It can be stated that the slicing/chunking approach decreases complexity to a much greater extent when specifications are getting larger. In fact, it can be shown that the mean value of the increase of complexity of the generated fragment is definitely less than the increase of the size of the specification.

5. Conclusion

Based on the argument that formal specifications are useful not only during initial software development but also during maintenance, this work presents specification fragments to help maintainers obtaining a focussed understanding of specifications.

More than 600 specifications fragments have been examined in order to demonstrate the efficiency of the approach. The experiments demonstrates that the approach can be used to achieve the goal of scaling down specifications in order to make them more comprehensible. It is shown that the generation of specifications fragments is sufficiently efficient.

The specifications used in this work represent typical specifications to be found in the Z-literature. However, much larger specifications (with several thousands of primes) are still waiting to be examined. It is likely that the approach still proves useful. The results generated by the prototype imply that, with larger specifications, there is generally a trend toward higher extents of reduction of complexity.

References

[1] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software Systems*. John Wiley & Sons, Inc., 1995.

[2] A. Bollin and R. 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.

[3] A. Bollin. Specification transformation as a basis for specification comprehension. In *Proceedings of Applied Informatics 02*. AACE, 2002.

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

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

[6] I. Burnstein, K. Roberson, F. Saner, A. Mirza, and A. 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.

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

[8] D. Jackson. Structuring Z Specifications with Views. *ACM Trans. on Software Engineering and Methodology*, 4(4), October 1995.

[9] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, 2nd edition, 1990.

[10] T. J. McCabe. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1989.

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

[12] T. Oda and K. Araki. Specification slicing in a formal methods software development. In 17th *Annual International Computer Software and Applications Conference*, IEEE Computer Socienty Press, pages 313–319, November 1993.

[13] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *10th International Workshop on Program Comprehension (IWPC'02)*, June 2002.

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

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

[16] F. Tip. A Survey of Program Slicing Techniques. Technical report, CWI Netherlands, 1994.

[17] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE, 1982.

[18] J. Zhao, J. Cheng, and K. 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.