Maintaining Formal Specifications – Decomposition of large Z-Specifications

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

Abstract

As part of different maintenance models formal specifications can act as valid artifacts for maintenance tasks. However, the linguistic density of specification languages and the size of specifications might still be seen as an obstacle against comprehension, reuse, and change activities.
This paper introduces an approach for the identification of specification fragments of Z specifications with a well defined semantic content. These fragments, namely specification chunks and specification slices, not only support comprehension tasks, they also enable maintenance personnel to identify and focus on the relevant parts of specifications for the problem at hand. Their ease in creation and use makes them well suited for maintenance, as is demonstrated by a simple prototype for Z specifications.

1. INTRODUCTION

Failure and success of formal methods depend on the viewpoint (and the person) from which (from whom) they are going to be assessed. There are studies and experience reports that tell about benefits in areas like railway and Metro systems, telecommunication, and other security related fields of application [10, 17, 19]. On the other it often is argued that existing formal specification environments are not practicable at the moment [12]. So how can formal specifications then play a crucial role during maintenance phases? A commonly known argument is: "Our program code exactly describes the behavior of the implemented system, why not throwing away the specification/design documents and just taking the underlying software code as the sole input?"
There are at least three aspects that are easily forgotten:

- Reconstructing all concepts from program code is a time-consuming task. It requires a lot of tool support which ultimately means the reconstruction of the missing documentation at different levels of abstraction.
- Specifications provide a trustful source for the description of the original requirements, especially when they are kept up to date during the 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.

These arguments suggest that it pays off to deal with formal specifications in maintenance phases. However, the formal methods community still has to face the fact that formal specifications are not really state of the practice.

The benefit of a formal specification as a useful source for documentation, test, and refinement is true if and only if the specification is kept up to date during evolutionary steps taking place during development, and later, during operation and maintenance. To keep formal specifications up to date requires effort, and this effort obviously is meant to be too high. This is supposed already for small specifications and with growing sizes of specifications the situation even gets worse. The specification of the air traffic control system (CDIS, Central Control Function Display Information System [13]) had about 1000 pages, and managers often claimed that formal specification languages are merely "write-only" languages which do not scale up.

The problem is not new. Programs as well as specifications get complex if their size exceeds some limit. The good news is that for programs there are a number of approaches supporting maintenance tasks. The bad news is that there is no (tool) support for maintenance tasks concerning formal specifications. As maintainability of a system depends on the ease of maintaining the underlying system and documents, and as specifications are not that well supported, it is no wonder that formal specifications have not found their way into industrial practice.

This paper suggests an approach that eases comprehension and maintenance tasks of specifications. It enables the maintenance personnel to focus on those parts of the specification that are relevant for the problem at hand. Section 2 starts with an overview of the context of the maintenance tasks and discusses the state of the art but also the problems one is running into when maintaining formal specifications. Section 3 presents the general idea of how to deal with complex formal specifications. Section 4 introduces a simple prototype for Z specifications which has been designed to support the maintenance personnel and discusses its field of application. The paper concludes with a summary of recent findings and an outlook of how the approach is going to be improved.

2. MAINTAINING SPECIFICATIONS

The average lifetime of a software system is said to be about a decade. It thus is no wonder that more than half of the resources spent on the system is spent for maintenance activities [2]. These activities are not only influenced by the type of maintenance, they are highly influenced by the artifacts of the system available to the maintenance staff. These artifacts, their level of abstraction, and the underlying maintenance process model define the maintenance context.

2.1 The Maintenance Context

In [18] Pirker identified four contexts for software maintenance. His focus was on product composition, and depending on which artifacts are available at hand, he differs between classical systems and truly maintainable systems. When only the binaries are available this is referred to as the *pathological maintenance* context. When the source code is available, too, the situation is regarded to as the *classical maintenance* context. The *model-based maintenance* context already includes design and/or specification models. Finally, when additional clues (so called hooks) are available, the context is that of a *maintainable system*.

Of the four contexts the last two are most interesting for our considerations. Specification documents are available and can support maintenance activities. However, they are used slightly differently depending on the underlying maintenance process model [1, 14]. Among them there are:

- The *Quick Fix Model* which focuses on code. Usually there is a change request on the code. After compilation and test this change influences all other artifacts, including the specification. In this situation it is important to *identify related parts within* the specification.
- The *Iterative Enhancement Model* might be compared to prototyping approaches. It starts with the analysis of the old system, which, in consequence, leads to a set of modified requirements. The whole life-cycle is repeated. Here, the specification is first used as an input source for *comprehension activities*, thus used to provide *concepts* to the maintenance personnel. Afterwards the specification is adopted to the changing requirements.
- The *Full Reuse Model* is based on the reuse of parts or components of the old system. The maintenance staff has to decide, which parts of the system are to be reused, and which are not. Here the specification, as an artefact, is analyzed for reuse, too. It is important to *carve out components* of the specification in order to decide whether they are suitable for reuse or not.
- The *Specification Based Maintenance Model* focuses on changing the specification and design documents before starting typical refinement steps. Here it is necessary to rapidly *identify dependent parts within* the specification (as they are likely to be influenced by a planned change, too).

Not only the models can be mixed up, the different activities of dealing with the specification are also interrelated. The requirements for an approach supporting specification maintenance are manifold: it should be possible to identify related and/or somehow dependent parts of the specification, and it should be possible to carve out components, or concepts. To summarize, maintenance support consists of support for

- analysis of change and ripple effects,
- design recovery, and
- restructuring of the formal specification.

Solutions are provided in the field of specification comprehension and will be presented in chapter 3.2.

2.2 Impediments

Specifications may contain several thousand lines of specification text, and the deducted software system might consist of millions of lines of code. Thus, often the set of overwhelming details of a specification cannot be understood by a single person anymore. The artifacts are said to be ”complex”, and as in most cases we do not have the above mentioned truly maintainable system, we have to deal with typical comprehension activities. Up to this there is no difference between maintaining programs and maintaining specifications.

In any situation the maintenance personnel has to fully comprehend the relevant parts of the system, and *size* is the most influencing (and limiting) criteria when dealing with the artifacts. But it is not only the size that matters.
As is argued by Mittermeir and Bollin in [15], 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.

Edmonds introduces the term *analytical complexity* [11] for the type of difficulty when trying to comprehend somebodies expressions. There are several reasons why the reconstruction of the original concepts behind a formal specification is that hard. Reasons are:

- Missing Redundancy. The density of expressions has an important impact on various 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.
- 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 in general have no built-in methodologies for structuring. Some notations provide abstraction techniques on the granularity level of objects, but they are of no use when specifications are getting really big.
- 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 understand-

ing 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.

Missing redundancy 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.

It is the last reason (missing clues for behavior and structure) that causes the effort and the experience of complexity. A lot of solutions have been provided in the field of program comprehension dealing with such types of complexity. They are mainly based on the identification of control and/or data flow in the program. This information is then used to carve out parts of the program, to visualize the program, and to identify dependent parts.
However, due to the above mentioned differences (no execution order, no direct flow of control) typical program comprehension approaches cannot be applied directly to formal specifications. In [4] the notions of control and data dependency (between well-defined parts of a specification) have been introduced, and this paper explains how this approach can be used in order to support the above mentioned maintenance activities.

3. MAINTENANCE SUPPORT

As argued in [15], the complexity of specifications is the main reason why developers shy away from using specifications. The density of expressing thoughts in specifications becomes detrimental for comprehending them during later phases. This section presents an approach of how to deal with the complexity of specifications and how this findings can be used in order to ease maintenance tasks.

3.1 Dealing with Complexity

Formal specifications remain compact structures, but in respect to maintenance and comprehension tasks the complexity of specifications can be reduced effectively. The key idea is to support the maintenance personnel with well-defined types of specification abstractions, namely *partial specifications*:

- A partial specifications is smaller than the original specification, but contains all relevant parts of interest (for the problem at hand).
- When partial specifications are substantially smaller than the full specification, they are easier to grasp. It is size that matters.
- In an optimal case partial specifications are derived from the full specification automatically.

The informatics literature contains several concepts of partiality, aiming to provide an interested party just the perspective needed for a particular task. Especially the notions of slices and chunks come to mind. In the following the concept of partial specifications is discussed in more detail.

3.2 Partiality

There is definitely a need for partial specifications. As will be explained in the subsequent section for Z specifications, slices and chunks appear to be promising candidates as they are decomposing the specification in a well-defined manner.

At the first glance various types of partial specifications (sometimes also called specification abstractions) are of interest. However, looking only at terminals (literals) of the specification language is not sufficient, and defining arbitrary sets of literals as units is not practicable either. Semantically meaningful units are needed.
For the definition of suitable elements a bottom-up approach has been chosen:

Specification Literals

In general, specifications are constructed from basic (atomic) units. These basic elements are called *specification literals*. They can easily be identified by looking at the grammar of the specification language. As an example, specification literals can be keywords of the specification language, any operators or identifiers. When looking at the Z set-comprehension expression

$$\{x : \mathbb{N} \mid x < 5\}$$

the set of specification literals is $\{\ '\{', 'x', ' :', '\mathbb{N}', ' |', ' <', '5', '\}'\ \}$. However, specification literals are not very expressive when standing alone. It is the combination of literals that makes them rich in content.

Prime Objects

By aggregating specification literals, *prime objects* of a specification can be built. In specification languages these prime objects can be expressions, predicates, or even generic type defintitions or schema type definitions. Some examples of Z specification primes are:

$$\mathit{Report} ::= \mathit{OK} \mid \mathit{NOK}$$
$$\mathit{result}! : \mathit{Report}$$

or

$$[\mathit{limit} : \mathbb{N} \mid \mathit{limit} = 10]$$

Prime objects are not restricted to simple expressions. As they form logical units, the simple primes mentioned above can be combined together in order to form so-called higher-level primes.

Definition 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.*

The following 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.

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.

Specification Fragments

The assembly of several specification prime objects leads to specification fragments.

A *specification fragment* consists of several prime objects, but does not necessarily constitute a complete specification. It is a composition of several primes which are isolated from their surrounding context. The following set of primes forms a simple specification fragment:

Add

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

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

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

This 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. Based on syntactic elements, several types of abstractions can be derived: chunks, slices and clichés form so-called *semantic elements* of a specification.

Specification Chunks

Chunks are syntactic or semantic abstractions of text structures. In accordance with the definitions provided in [8], 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 2 *A* **specification chunk** *is*
(i) a prime including all primes contained within it, or,
(ii) a set of primes that exists within the same specification scope. For each pair of primes within the set of primes either one prime is dependent on the other or both primes are dependent on a third prime (within the set of primes).

The really important thing is that a chunk always has to be comprehensible in isolation. Compared to a specification fragment, a chunk contains enough (surrounding) context to stay understandable. That means that at least enough semantic information has to be taken into consideration when generating specification chunks.

The following Z specification is derived from a popular example in the formal methods literature [20]. It is the specification of a simple database called birthday book (*BB*, see App. A for the full specification) written in its horizontal form. It allows to store, search for and delete names and dates of birth.

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

Based on the above birthday book specification, a specification chunk can be generated by looking at the specification prime $birthday' = birthday \cup \{name? \mapsto date?\}$ in the *Add* operation schema and by regarding all primes which are data dependent on that prime and which are necessary to ensure the correct syntactical context:

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

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 section 3.3.

Specification Slices

Slices have already been defined for specifications. In 1993, Oda and Araki [16] first used static slicing techniques for analyzing Z specifications based on a simple definition of data-dependency. One year later, Chang and Richardson [9] 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.

Chang and Richardson define their slicing approach in a top down manner. The limits of their definitions are multifaceted and are discussed in more detail in [4]. The definition of a slicing function is too general and had to be re-stated. On the other hand the definition of their slicing criterion for specifications is too Z-related and thus has to be re-stated, too. Instead of "lines", prime objects were taken as basic, structuring units.

Definition 3 *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 *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 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:

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

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) would make the process much more difficult. A function that deletes "the parts not needed" then has to be applied to the specification. Either the word "needed" gets very complex semantics, or the semantics of the fragmental specification cannot necessarily be given.

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 [7]. As for program clichés, the following definition can be provided for specifications:

Definition 5 *A* **specification cliché** *is a basic knowledge unit used to build and recognize specification text.*

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.

3.3 Hidden Dependencies

As mentioned above, declarative specification languages do not provide an explicit notion of control. On the other side, the definition of control is based on the concept, that a statement is evaluated. This evaluation then decides whether another statement is executed or not. A similar concept can be identified within specifications: pre- and post-conditions. The pre-condition part is evaluated and this evaluation determines whether the post-condition part of the specification is applicable or not [4].

The following Add schema of the birthday book specification contains two predicates. The first predicate checks whether the provided name is in the birthday database. The second predicate contains an after-state identifier ($birthday'$) and adds the name and the accompanying date to the database of birthdays. These two predicates can be interpreted as a *pre-condition prime* and a *post-condition prime*.

$$\begin{aligned} Add == [&\Delta BB;\ name? : NAME;\ date? : DATE \mid \\ &name? \notin known; \\ &birthday' = birthday \cup \{name? \mapsto date?\}] \end{aligned}$$

The crux of the matter is that a semantic analysis of the pre and post-conditions does not generally lead to the above primes. The situation gets worse when Z schemata are combined by using Z schema operators like composition ($\fatsemi$) or implication ($\Rightarrow$). In [4] Bollin introduced the notion of a syntactical approximation to the semantic analysis and ended up with a simple set of rules for the identification of relevant prime candidates. They are summarized in Tab. 1. E.g. when two schemata S and T are combined by using sequential composition ($\fatsemi$), then there is control dependency ($\rightrightarrows_c$) between post-condition primes in S (po_S) and T (po_T) and the pre-condition prime of S (pr_S).

Schema Operation	Related Primes
S	$po_S \rightrightarrows_c pr_S$
$\neg S$	$po_S \rightrightarrows_c pr_S$
$S \vee T$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \Rightarrow T$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \wedge T$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \Leftrightarrow T$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \upharpoonright T$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \fatsemi T$	$(po_S \cup po_T) \rightrightarrows_c pr_S$
$S \gg T$	$(po_S \cup po_T) \rightrightarrows_c pr_S$

Table 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. *po* **denotes primes containing after-state and output identifiers,** *pr* **denotes primes containing no after-state identifiers.**

The following definition for the identification of control dependencies within schema boxes can be advanced:

Definition 6 Control dependencies in Z schemata. *Let S be a schema of a syntactically correct specification. Furthermore, let pr_S be the non-empty set of pre-condition primes of S and po_S the non-empty set of post-condition primes of S. Then primes in po_S are said to be control dependent on primes in pr_S (abbreviated as $po_S \rightrightarrows_c pr_S$).*
When Z schemata are combined using Z-schema operators, then primes in po_S are said to be control dependent on primes in pr_S in respect to the rules presented in Table 1.

With the notion of control, the definition of data dependency gets possible, too. The parts to look for are the definition (or assignment) of values and the use of data elements. In Z, a literal denoting a data element is said to be an identifier. According to the terminology used in the Z community, an identifier is said to be *declared*, if it appears at the left side of a declaration or at the left side of a schema expression. It is said to be *defined*, if the identifier is decorated and appears at the left side of a value assignment. It is said to be *used*, if it is neither declared nor defined.
Based on this terminology, data-dependency in Z specifications is defined as follows:

Definition 7 Data dependency between Z-primes. *A specification prime p is* data dependent *on a specification prime q $(p \neq q)$ if*
(i) there exists at least one identifier v (literal denoting a data element) that occurs in both p and q, and
(ii) v is defined in q and used in p, and
(iii) either p and q are in the same scope, or p is control dependent on q.

With the introduction of the notion of control in Z specifications, it gets possible to define control and data dependencies within state-based specifications. However, it is just a means to an end. Up to now several forms of abstractions have been

discussed. Not all abstractions are useful in all situations. Their applicability depends on the problem at hand and thus on the point of interest.

3.4 Applicability

Section 2.1 already discussed the different situations in which formal specifications might be involved during maintenance tasks. The previous section introduced the notions of prime objects and dependencies, and in the sequel the applicability of the approach is discussed. It refers to the requirements for tool supporting specification maintenance.

- *Analysis of change and ripple effects*. A requirement changes and thus a small part (e.g. a prime) of the specification is about to change, too. It gets necessary to see the minimal portion of the specification affected by 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. This situation can be controlled by generating specification *chunks*. On the other hand both specific and distributed portions (fragments) of the specification text might be of interest. This situation could arise when looking for operations that modify a well defined set of state variables.
- *Restructuring*. Again, slicing and chunking techniques can be applied when identifying and changing whole portions of the specification text. A chunk (or set of chunks) helps focusing on specific parts of the specification, and, whenever the focus is clear, the slice guarantees that all relevant portions of specification text can be considered.

The above mentioned maintenance tasks can all be sustained by slices, chunks, and related fragments. This does not mean that other approaches are to be excluded, but with partial specifications two important activities become possible: focusing on *small parts* of the specification and putting an eye on *relevant* (which means *dependent*) parts of the specification. With slices, chunks, and fragments, the basis for useful abstraction is provided.

Remains the question of how to focus on the point of interest. As with program slicing it is the abstractions criterion that influences the result, and in analogy to a program's slicing criterion, the criterion for specifications consists of several parts:

- Firstly, the focus will have to be set to a specific position in the specification. As argued in section 2.1 it only makes sense to look at the smallest entity with a well defined semantics available in a specification: the specification *prime*.
- Secondly, different types of abstraction require an adjustable focus. The focus itself can be set by making use of two features: in the first place there are the types of *dependencies* that are of interest. However, adjusting the focus via inclusion or exclusion of dependent primes is a rather coarse mechanism. Thus, the focus should additionally be adjusted by considering specification *literals*.

Thus the criterion for creating specification abstractions consists of three parts: a specification *prime* (representing the point of interest), a description of relevant *dependencies* (that have to be considered) and a set of *literals* (for optionally fine-tuning the selected dependencies).

Based on the definitions of primes and dependencies, slices and chunks can be generated automatically from a given specification. All that is necessary is to provide the appropriate abstraction criterion. Section 4 introduces a simple prototype that is able to support the above mentioned maintenance activities for Z specifications.

4. A PROTOTYPE FOR MAINTENANCE SUPPORT

In order to deal with large specifications, a simple text-based prototype for Z has been implemented. It serves as a basis for the experiment presented in the remainder of this section. This section describes the prototype in more detail.

4.1 Technical Background

The following tasks can be handled by the prototype:

- It identifies dependencies hidden in the specification.
- It calculates partial specifications, that is chunks and specification slices.
- It transforms the specification to a net and visualizes the specification such that primes are represented as vertices and dependencies are represented as arcs.
- It calculates statistics based on the net representation of the specification. This includes the number of primes, the number of control-dependencies, and the number of data-dependencies.
- It generates output in two ways. Firstly, the net can be stored in a graphical format in order to visualize the net via *dotty*. Secondly, the net can be transformed backward to the specification source.

To ensure portability, the prototype (in the following called "*SliZe*" toolkit) has been implemented in Java. It is available for Windows and Linux platforms. For reasons of simplicity, the *SliZe* toolkit is based on the *preccx* grammar[1] of Z which has been defined by Breuer and Bowen [6]. The compiler produced by *preccx* is able to check for syntactically correct Z specifications written in LaTeX and has been modified in order to produce an intermediate representation of the specification. This representation then serves as an input to the *SliZe* application (see Fig. 1).

[1] Breuer and Bowen's *preccx* home-page: http://www.afm.lsbu.ac.uk/archive/redo/precc.html. Page last visited: March 2004.

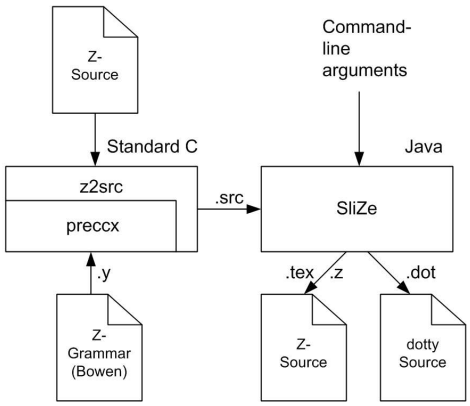

Figure 1. General structure of the prototype. The *preccx* grammar of Z is used to generate an intermediate representation of the specification. This representation serves as an input to the *SliZe* application. The application itself is controlled by command-line arguments and enables both, the generation of partial specifications and the generation of *dotty* representations of the net.

There have been 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):

- The aim is to make the approach (and 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.
- The *preccx*-grammar, as defined by Breuer and Bowen, is the only officially available grammar for Z-specifications which are type-set in LaTeX. It is based on the language definition of Spivey [20] 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[2] and CUP[3]) 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*[4]).

The prototype has some limitations when displaying and interacting with the net. As dotty tries to optimize the layout of the net, large specifications lead to very high computation times. A solution would be to abstract from the details in the net and to provide some sort of fish-eye views. Another limitation is that the prototype only provides a text-based interface.

[2]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.
[3]CUP home-page (Scott E. Hudson):
http://www.cs.princeton.edu/ appel/modern/java/CUP. Page last visited: March 2004.
[4]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 [6] for more details.

Name	#P	V	A	CC	v'(l)	v'(u)	DU
BB	1	72	267	24	4	10	4
Petrol	3	134	674	53	10	28	131
Elevator	6	349	3668	144	32	1096	1212
WM	12	520	2644	213	39	544	215

Table 2. Complexity overview regarding four Z-specifications. The table summarizes the number of pages of specification text $\#P$, the total number of vertices V and arcs A in the net representation, the conceptual complexity CC, the extended cyclomatic complexity $v' = (v'(l), v'(u))$ and the DU count metric.

Despite these limitations, the prototype proves to be useful. It simplifies the analysis especially when generating different partial specifications. In the context of an experiment more then 600 partial specifications have been generated and thousands of dependencies have been detected. And all this within a few minutes.

4.2 Experiment

In order to compare the specifications in respect to their complexity, the following measures are introduced [4]:

- Conceptual Complexity $CC(\Psi)$ of a specification Ψ. This measure is based on the number of basic prime objects in the specification (and not on the unprecise 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 the additional definition of the number of vertices and arcs in the net presentation, a wide range of specifications' complexity measures exists. A comparisons between different specifications become feasible.

The birthday book specification is too small to benefit from the suggested approach. For that reason three other specifications have been looked at. The "Elevator" specification [9], the "Petrol-Station" specification (*Petrol* for short) which was used during class labs at the University of Klagenfurt and the "ITC Window Manager"-specification (*WM* for short) which is a commercial specification presented in [5]. Table 2 summarizes the complexity of the specifications and the complexity of the net representation.

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,

Specification	*Slice*	$Chunk_1$	$Chunk_2$	Mean
BB	0.80	0.73	0.81	0.78
Petrol	0.83	0.73	0.48	0.68
Elevator	0.86	0.33	0.22	0.47
WM	0.64	0.32	0.32	0.43

Table 3. Reduction factor k ($= Size_{new}/Size_{old}$) for four different specifications and the generation of slices and two different types of chunks.

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

The specifications have been used to investigate the question whether and to what extent the generation of partial specifications pays off. The BB, Petrol, Elevator and WM specifications were used as experimental objects for the treatment: the application of the generation of partial specifications. The prototype mentioned in section 4.1 has been used to generate all possible sets of partial specifications for every predicate prime in the specification. These primes acted as the points of interest[5]
For every point of interest three different types of partial specifications were generated. For every prime vertex a full static specification slice, a full static specification chunk focusing on data-dependency, and a full static specification chunk focusing control dependency were calculated.

The experiment, particularly the efficiency of the generated partial specifications, is discussed in more detail in [3]. The subsequent section summarizes the two most important findings in respect to maintenance tasks.

4.3 Results

By automating the generation of slices and chunks it is possible to reduce the time for detecting the minimal portion of the specification to be changed to a few seconds. Once the specification is analyzed by the *sliZe* toolkit it is not necessary to look for hidden control and data dependencies within the specification text by hand anymore. Especially the generation of chunks leads to a drastic reduction of size, and with it to a drastic reduction of time needed to comprehend the partial specification. In respect to size complexity the prototype definitely prove useful.

The experiment demonstrated that the effect obtainable by slicing and chunking rises with the size of the specification under consideration. For the mean values of the above introduces complexity measures this observation had been confirmed (see table 3). Furthermore it could be stated that the slicing/chunking approach decreases complexity to a much greater extent when specifications are getting larger. It was also shown that the mean value of the increase of complexity of the generated partial specification is definitely less than the increase of the size of the specification.
These findings encourage the use of the approach for specification maintenance activities.

[5] 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.

5. CONCLUSION

Specifications are valid sources in different maintenance process models and lead to an increase in maintainability. However, when there are no built-in hooks complexity hinders typical maintenance and comprehension tasks. This paper discusses typical impediments, but also elaborates on the different situations when specifications prove useful.

Formal specifications remain compact structures. However, in respect to maintenance and comprehension tasks the complexity of specifications can be reduced effectively. This can be achieved by focusing on those parts which are necessary to solve a specific problem at hand. Several factors contribute to the overall complexity, but the vast majority of problems goes back to the complexity of size; thus this work presents an approach to reduce the size of specifications. It is suggested to generate well-defined partial specifications such as specification slices and specification chunks.

The paper also introduces a prototype that has been written to sustain the maintenance of Z specification. The prototype will have to be improved to better incorporate into existing working environments – especially the lack of a graphical user interface has to be eliminated. However, the small experiment shows that the prototype is useful and that the generation of partial specifications does make sense in respect to maintaining formal specifications.

REFERENCES

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

[2] K. H. Bennet. *Software Maintenance: A Tutorial. In M. Dorfman and R. H. Thayer, Software Engineering*, pages 289–303. IEEE Computer Society Press, 1997.

[3] A. Bollin. The efficiency of specification fragments. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering*, 2004.

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

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

[6] P. T. Breuer and J. P. Bowen. A concrete Z grammar. Technical Report PRG-TR-22-95, Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK, 1995.

[7] A. Broad and N. 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.

[8] 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.

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

[10] E. M. Clarke and J. 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.

[11] B. Edmonds. *Syntactic Measures of Complexity*. PhD thesis, University of Manchester, 1999.

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

[13] J. A. Hall. Using formal methods to develop an atc information system. *IEEE Software*, pages 66–76, March 1996.

[14] K. Lano and H. Haughton. A specification-based approach to maintenance. *Journal of Software Maintenance: Research and Practice*, 3:193–213, 1991.

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

[16] T. Oda and K. 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.

[17] S. L. Pfleeger and L. Hatton. Investigating the influence of formal methods. *IEEE Computer*, 30(2):33–43, Feb. 1997.

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

[19] A. E. K. Sobel and M. R. Clarkson. Formal methods application: An empirical tale of software development. *IEEE Transaction on Software Engineering*, 28(3):308–320, March 2002.

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

APPENDIX A - BIRTHDAY BOOK SPECIFICATION

The birthday book (BB for short) describes a simple system for administrating names and birthday dates.

First names and dates are introduced as global sets. In order to indicate the success or failure of an operation, a global type *REPORT* is introduced.

$$[NAME, DATE]$$
$$REPORT ::= OK \mid NOK$$

The state space consists of the set of all known names, and the "database" entries for the birthday dates. The predicate ensures that only known names are in the database.

BB

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{dom}\, birthday$$

At the beginning the database is empty.

InitBB

$$BB$$
$$known = \varnothing$$

There are several operations for working with the database. It is possible to *Add* a pair $(name, date)$ to the database, it is possible to *Delete* an entry from the database, and to *Find* a birthday date in the database.

Add

$$\Delta BB$$
$$name? : NAME$$
$$date? : DATE$$
$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

Delete

$$\Delta BB$$
$$name? : NAME$$
$$name? \in known$$
$$birthday' = birthday \setminus \{name? \mapsto birthday(name?)\}$$

Find

$$\Xi BB$$
$$name? : NAME$$
$$date! : DATE$$
$$name? \in known$$
$$date! = birthday(name?)$$

To indicate the success of an operation the result *OK* is returned.

Success

$$result! : REPORT$$
$$result! = OK$$

With the above operation schemata the functioning system consists of successfully performed add or delete operations.

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