Specification Transformation as a Basis for Specification Comprehension

Andreas Bollin
Institute for Informatics-Systems
Klagenfurt University
Universitätsstrasse 65-67
A-9020, Austria
Andreas.Bollin@uni-klu.ac.at

ABSTRACT
This positioning-paper explores the question of whether program slicing and analyzing techniques can be applied to the field of specification comprehension. The scope of visualization tools and techniques as well as the strategies that programmers use to comprehend programs are varying widely - and these techniques and theories seem to hold also in the field of specifications. Indeed, specification transformations, in combination with specification visualization techniques, substantially aid the comprehension process. This paper introduces the field of specification slicing and elaborates the problem of finding suitable dependencies, as a basis for the above mentioned specification transformations.

KEY WORDS
Specification comprehension, slices, specification visualization, dependencies

1. Introduction

Whether it is a myth and/or a fact, formal specifications [17, p. 2303] are often criticized to be hard to understand. Criticism [5] includes:

1. Mathematical constructs and operators are requiring paradigms expertise.

2. There is a lacking connection to other representational forms, regardless if upstream or downstream in the software development process, thus specification expertise is needed.

3. Last, but not least, specifications might get large, they are containing too much information, and hence the reader is overwhelmed due to the semantical compactness of the specification.

Specifications cannot be treated like programs, but the problems are similar when trying to understand them. Program slicing techniques are often used for focusing on relevant parts of the program, but as we will see in the second chapter of this paper, there are significant differences between program dependencies and specification dependencies - applying existing tools (like the Program Dependence Analysis Graph System ProDAG [10]) onto specification will not really work. Another approach is the use of software visualization tools in order to make programs more readable - and as specification visualization is related to program visualization (at least at the pragmatics level of visualizing), there is realistic hope, that techniques and tools for program visualization can be used for specification visualization, too.

The following section tries to close the gap between the well-known field of software comprehension and specification comprehension and in the remainder of the papers the idea of specification transformations is introduced in order to provide techniques to the reader of specifications similarly to that of program slicing.

2. Program and Specification Comprehension

In the late 1980s models have been built, helping to understand how programmers comprehend programs. The so called ”mental model” describes the programmers mental representation of a program (or specification) to be understood [11]. The ”cognitive model” describes the information structures and processes used to form that mental model. For many years studies observed how programmers understand programs, and cognitive models for program comprehension have been proposed. It is an advantage, that all models have a lot in common, particularly they use existing knowledge to acquire new knowledge and to create a mental representation of the code. Furthermore they are using matching processes between the (already known) knowledge structures and the program under study - and by knowing about these structures (and abstraction objects) we are able to build tools to aid the comprehension process.

Models like the bottom-up and top-down model are not restricted to just programs, all of the cognition models are also valid for specifications. Tilley [15] describes three basic activities typical for comprehension tools, and these activities can again be mapped into the field of specification comprehension: the first activity is data gathering, using (static) analysis of code as well as using (dynamic) analysis based on the execution of the program. The second activity comprises knowledge organization by creating abstraction objects. The last activity supports information exploration by providing navigation aids, analysis tools and tools for representation.

The RIGI reverse engineering tool [8, 14] is an excellent example for such a kind of tools as it provides so-called views to direct the user's focus on visual and spatial data in order to support program documentation and understanding.

When taking a closer look at tools and techniques for formal methods, one will figure out, that there are basic approaches supporting the comprehension process, too. As is explained in more detail in [7], specification visualization and comprehension techniques can be, at least, divided into three classes. The first class contains tools for writing, reading and browsing through specifications. The second class contains animation tools, and the third class describes tools that are a combination of (parts of) the above two classes and rewriting tools in order to support reasoning and analyzing processes.

Animation and execution seem to be fruitful techniques in raising comprehensibility, but most of the specifications are not directly executable - and that is the limiting factor. By using libraries to transform the specification into an executable programming language (often Prolog) it is at least possible to animate subsets of the specification. Secondly, rewriting and reasoning tools are also promising, but they are merely for experts use. Taking different comprehension models into account, one thing seems to be clear - grouping "related" parts together, and aiding the selection and abstraction process by providing only relevant parts (fragments, slices) to the reader is definitely supporting the understandability of specifications. As an example, one approach to ease the comprehension of specification is the use of different views onto the source itself. Daniel Jackson [6] introduced partial specifications (consisting of a state space and a set of operations), and composes these views into a full specification. When using multiple representations (different views) it is possible to improve the clarity and modularization of specifications. In order to generate interesting abstraction objects, the use of techniques analogue to slicing techniques seems to be very promising. In the following section, this idea of specification transformation will be presented in more detail.

3. Specification Transformations

One reason for applying comprehension techniques to a specification is to weaken their semantical compactness, thus reducing the "size" of the specification, splitting it into smaller, more readable and understandable parts. But is it possible and reasonable to split a specification, is it possible to build clusters, and partitions? Actually, (static) slicing techniques can help to answer this question [9]. In fact, there are many facets that have to be concerned when talking about the support of specification comprehension.

3.1 Specification Comprehension

Taking a closer look at this problem, one might find out, that there are again two related strategies to be concerned:

1. partitioning, with the size as the matter to deal with (thus trying to reduce the syntactical complexity), and

2. visualization, in order to support the abstraction process (thus trying to reduce the semantical complexity).

For either strategies it is important to identify only the relevant parts of the specification - and the questions are: how to find those parts, and which parts are relevant? Slicing techniques, as introduced in [4] by Chang and Richardson can help, but the fact, that the resulting specification (in order to support, to guide, and not to confuse the reader) has to stay syntactically and semantically correct, should not be neglected. Semantical correctness, at least in the context of the focus the reader has in mind, will be as important as identifying the relevant parts.

The questions, one might have in mind when reading a specification, are often rather simple. When taking the specification language Z [13] as an example, we might want to know, which operations, under which conditions, are modifying the state space. On the other hand, there are many complex questions:

> *Which schemata are relevant for a specific series of operations? Under which conditions are specific schemata relevant? Is there a predicate that is never true?*

The first and second question might arise, if one wants to understand the behaviour of a specific series of Z operations, the third question tries to identify parts of the specification, that are not relevant at all. There exists also a second class of questions relevant in that context. In order to support readability, it might be good to know, whether it is possible to split a specification into several parts. Smaller parts aid the comprehension process, but the possibility to split a specification might indicate a poor specification quality [12, 18]. To summarize, all questions might occur during the comprehension process, but also during debugging, which is another important aspect in that context.

Specification slicing is a valid technique in order to generate relevant sub-specification, but in many cases the sub-specification (as it is the same in the field of program slicing) is nearly as large as the original specification. Other transformation and partitioning techniques will have to be explored in order to reach our high-level goal of abstracting from unnecessary details.

3.2 Slicing Specifications

Depending on the abstraction objects the reader is intending to use, the above mentioned sub-specification might turn out to be views [6], slices [9], chunks [3] , fragments, or, simply, partial specifications. Following the definitions in the field of program comprehension [11], chunks are syntactic or semantic abstractions of text structures within the source code. Those chunks can be collected and abstracted further, in order to build higher-level chunks. A specification fragment is an incomplete or isolated portion of specification code and a partial specification is specification fragment, representing a state space and a set of operations. All of these abstractions are helpful, but different, and therefor affect the application and type of our slicing (transformation) functions.

It is important to note, that abstraction elements should not miss-lead the reader and suggest a meaning contrary to the meaning of the original specification. To aid the comprehension process, there will have to be support for identifying useful abstraction objects, but there will also have to be support for entering understandable questions and support for visualizing the results.

At the first sight, the idea of applying slicing functions to specifications seems to be nothing new or challenging. The simple idea is to map the definitions of slicing functions from program slicing into the world of specifications. But indeed, the mapping renders the challenging part. Programs and specifications are, at a second sight, different. In programs we have an explicit control flow, “line numbers” representing some kind of order, explicit control constructs. It is state-of-the-practice to identify control and data-dependencies. In the world of specifications we are not that lucky. There often exists no explicit control flow. In VDM [2] we have at least pre- and post-conditions guiding us in finding simple dependencies for functional units. In Z, as a second example, we do not have any syntactical notion of pre- and post-conditions, but at least pre- and post-conditions can be calculated. The same is valid for the definition of data-flow, and hence one has to be careful in putting specifications on a par with programs.

Nevertheless, slicing techniques seem to be a valid approach for aiding comprehensibility. So lets take a look at what is state-of-the-art in program and specification slicing. One remarkable overview can be found in Tip’s survey on program slicing techniques [16], where a collection of examples and algorithms is presented. In 1993, Oda and Araki [9] first used static slicing techniques for analyzing specifications, based on a simple definition of data-dependency. One year later, Chang and Richardson [4] introduced dynamic specification slicing, by extending the idea of Oda and Araki.

The original idea of slicing goes back to the PhD-thesis of Weiser [19], and is based on static data-flow analysis (flow-graph), allowing to find the slice in linear time as the transitive closure of a dependency graph. Beside the original definition of a slice (the slice has to remain syntactically and semantically correct) many different algorithms, even with weakened definitions of a syntactically correct slice, have been defined. Tip [16] defined a slice as a

> *... subset of statements and control predicates, that directly or indirectly affect the values computed at the criterion, but which do not constitute an executable program.*

This definition might be useful, but especially for specifications one has to be very careful not to destroy the semantic of the resulting abstraction object. The definition of a slicing function in the work of Chang and Richardson is similarly weak and is defined as follows [4]:

> *Any function removing tokens from the specification can be considered as a slicing function, as long as the specification remains syntactical and semantical correct.*”

Chang and Richardson are defining their slicing functions based on a simple definition of control and data dependency. The problem with their definition is multi-faceted:

1. First of all some slicing functions are not useful and it is important to identify those slicing functions that aid the comprehension process.

2. Secondly, the resulting slices, especially static ones, are not necessarily smaller than the original specification. Even when using dynamic slicing functions, the resulting sub-specification is not necessarily helpful, as the resulting specification is still not much smaller.

As mentioned above, the support of the comprehension process does not require slices only, but specification fragments and chunks are relevant, too - both are abstraction objects that are not assisted by their definitions. Chang and Richards definition of a slicing function is not general enough to support that requirement, thus it seems to make sense to search for further types of dependencies and slicing (or filtering) functions.

As mentioned above, it seems to be a problem to define useful slicing-functions. When taking a deeper look at the problem, it comes out, that the difficulties can be reduced to finding suitable dependencies in our specification. In order to identify and work with different dependencies it is useful to convert a given specification into a different type of representation (annotated graphs are a good and common target representation). The basic idea is, that the transformed representation can be analyzed and relevant parts of the transformed specification can be selected (filtered) more easily. Thus the transformation of our specification into another representation space - in Figure 1 the target space - enables the application of much simpler filter

Figure 1. Using transformations and filtering functions in order to support the comprehension process.

and selection functions. The resulting (sub-)specification then is transformed backward to the source space, leading to slices, chunks or fragments.

A second application of our transformation function is the animation (execution) of the specification (which is also a transformation from the source space on the left side of Figure 1 to the target and execution space on the right side). The filter function then is the specification's execution path, and the backward transformation will result in those parts of the specification that have been "touched" by the execution. For the sake of completeness it has to be mentioned, that there are some simple cases, where filter and selection functions in the source space are sufficient and a transformation into another representation space is not necessary. Imagine that one is just interested in the signatures of specified operations.

Another important aspect when dealing with objects of abstraction and slices is, that, (which is in contrary to the original definition) the object is rather constructed bottom-up and not top-down. A more useful definition of an specification abstraction (which could then be a slice) is the following:

> *A specification abstraction is a syntactically and (regarding to the context of the comprehension object) semantically correct specification, which is the result of the addition of those primes to an (initially empty) specification, that are directly or indirectly contributing to the abstraction criterion.*

Here a specification prime is, depending on the specification language (see [1]), a specification fragment, (in most cases a simple specification object, a specification predicate, a line, or a simple syntactically-glued part of the specification.

Nevertheless, it is still a problem to identify the primes (the basic elements of our specification) and the dependencies that are the basis for the contribution to the slicing criterion, so the next section introduces the field of specification dependencies and summarizes the findings of our latest research.

4. Specification Dependencies

As mentioned above, the original definition of a specification slice is based on a simple definition of data and control dependency. The definition of control dependency is kept very simple and is based on pre- and post-conditions analyses. As an example, in Z, a predicate q (here the prime object) is control dependent on a prime p, if p potentially decides whether q is applied or not. In the case of Z, the specification is split into parts of pre- and post-conditions and post-conditions are said to be control dependent on the pre-conditions. The definition of data-dependency is similar, a predicate (a prime) q is data-dependent on a predicate p, if data potentially propagates from p to q through a series of state changes. Based on these definitions and using a suitable slicing criterion, it was possible to define static and dynamic slicing functions for specifications.

In fact, this definitions in the work of Chang and Richardson are still very useful. Static slicing helps us to identify separate specification partitions and the idea of slicing can be implemented by the use of a simple annotated graph. Nevertheless, their approach does not directly support bottom-up comprehension or even the generation of specification chunks, partial and fragment specifications. When talking about that kind of abstractions, it is straight forward to widen the original definitions and to exert general transformation functions onto the specification in order to generate different types of specification abstractions.

The basic idea is to apply a specific transformation criterion (which is based on a pre-defined type of dependency) onto a transformed specification. The resulting specification then concludes those parts (primes) of the specification that satisfy the transformation criterion. The transformation criterion itself is a boolean function, deciding whether the prime object is satisfying a given predicate (the dependency) or not.

The problem then is not the definition of the transformation functions. The real challange is to reuse existing dependencies and/or to define useful dependencies. Dependencies in specifications can be explicit or implicit - more or less relying on the specification language one is using. Aside that, we identified three types of dependencies:

1. Primitive dependencies, including syntactical and structural dependencies. Every specification language consists of parts that are (notationally) related to each

other. In a Z state-schema, for example, at the top of the schema, there is a name, followed by the declaration part, and followed by the predicate's part below a horizontal line. These parts are belonging together, thus forming some kind of simple dependency. The position and the sequence of primes in the specification source also forms some kind of primitive dependency.

2. Static dependencies (relying on the primes and the static and logic relationship among them), In VDM, for example, pre- and post- condition parts can easily be identified, and basic control- and data-flow dependencies (static dependencies) can be defined.

3. Dynamic dependencies, also called semantic dependencies (where the meaning, the semantic of a prime object, is influencing the dependency). The content is relevant for calculating the relationship between specification primes. In Z, for example, dynamic dependencies can arise, when referring to included schematas and applying schema calculus. Another example would be the execution of a specification, the execution path then represents some kind of dynamic dependency.

With the dependencies and approaches identified in this section, it is possible to generate useful specification abstractions and support the building of the mental model in the comprehension process. Taking specification transformations as a basis, five approaches can help sustaining the comprehension process:

1. Generalization, via applying general transformation functions and general dependency analysis. The main idea is to weaken the semantical compactness of the specification.

2. Fragmentation and chunking via transformation functions, thus applying static dependency analysis. Here the main idea is to aid bottom-up comprehension processes.

3. Partitioning and Clustering, applying static (also called vertical) partitioning. The main idea is to reduce the compexity of size.

4. Compaction and reduction, applying dynamic and semantical (also called horizontal) partitioning. Here the main idea is to support top-down comprehension processes.

5. Extension, applying slicing and rewriting tools in order to change the level of abstraction.

As mentioned above, one problem to be solved is that of semantical correctness. Especially when dealing with dynamic dependency analysis the influence of semantical aspects cannot be neglected. The analysis of the influence of the filter functions onto the denotational semantic is of great importance and will have to be part of future research.

5. Outlook

As I have mentioned in the sections above, there is ongoing work in the field of software visualization and software comprehension. It is possible to combine the benefits of that research and the field of formal methods and by using program visualization, specification animation and transformation techniques, specifications will be, more understandable.

At the moment there exists no survey applying taxonomies to specification tools and approaches, but summarizing the findings of different comprehension models, and existing approaches, the following can be done to improve specification comprehension:

1. Pretty-printing, a technique that is already in use, can be used to ease systematically comprehension (as huge amounts of text have to be read) of specifications.

2. Specification transformations (in order to generate slices, chunks, partitions) can be used to help focusing on specific parts of the specification (as-needed comprehension). This approach directly supports bottom-up comprehension.

3. Different views onto the specification are possible in order to easy knowledge-based understanding. Static slicing an rewriting techniques will have to be used in order to generate different views.

4. And last but not least, fish-eye views, in combination with the above mentioned techniques, can be used to support top-down comprehension. Transformation functions are used to get a specific focus on the specification.

Again, all findings can be reused to overcome some of the problems, formal specifications are criticized for in the first section. According to the first problem, rewriter can help to understand complex mathematical constructs. Pretty-printers will raise readability and according to the second problem, tracers can establish the missing link between specifications and requirements and/or the implementation, raising the understandability of (parts) of the specification, providing different views onto the problem. According to the third problem, transformation tools can and will assist the reader in focusing on the parts of the specification of current interest.

Adding transformation functionality to existing tools (where data gathering, static analysis, presentation and storage are well supported) will lead to a toolset that is capable of handling all basic activities of comprehension tools (information exploration and abstraction) as demanded by Tilley [15].

6. Conclusion

This positioning-paper discussed approaches for raising comprehensibility of programs and tried to map the approaches and techniques onto the field of specification comprehension. At the moment there do not exist tools for the support of the cognition process, but the idea of using transformations, different views, tracers and rewriters seems to be very promising - in fact, specification comprehension will benefit from that research.

The combination of all the findings will, hopefully, lead to a toolset that assists users in comprehending specifications more easily ... and then it is really only a myth, that specifications are hard to understand.

References

[1] ALAGAR, V., AND PERIYASAMY, K. *Specification of Software Systems*. Springer, 1998.

[2] ANDREWS, D., AND INCE, D. *Practical Formal Methods with VDM*. McGRAW-HILL Book Company Europe, 1991.

[3] BURNSTEIN, I., ROBERSON, K., SANER, F., MIRZA, A., AND TUBAISHAT, A. 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* (November 1997), IEEE press.

[4] CHANG, J., AND RICHARDSON, D. J. Static and Dynamic Specification Slicing. Tech. rep., Department of Information and Computer Science, University of California, 1994.

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

[6] JACKSON, D. Structuring Z Specifications with Views. Tech. Rep. CMU-CS-94-126, Carnegie Mellon University, March 1994.

[7] MITTERMEIR, R. T., BOLLIN, A., POZEWAUNIG, H., AND RAUNER-REITHMAYER, D. Goal-driven combination of software comprehension approaches for component based development. In *23rd International Conference on Software Engineering (ICSE 2001)* (Toronto, May 2001), IEEE.

[8] MÜLLER, H., TILLEY, S., ORGUN, M., CORRIE, B., AND MADHAVJI, N. A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models. In *SIGSOFT'92: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments* (December 1992), vol. 17 of *5*, ACM Software Engineering Notes, pp. 88–98.

[9] ODA, T., AND ARAKI, K. Specification slicing in a formal methods software development. In *Seventeenth Annual International Computer Software and Applications Conference* (November 1993), IEEE Computer Socienty Press, pp. 313–319.

[10] RICHARDSON, D., O'MALLEY, T., MOORE, C., AND AHA, S. Developing and integrating prodag in the arcadia environment. In *In SIGSOFT '92: Proceedings of the Fifth Symposium on Software Development Environments* (December 1992), ACM SIGSOFT, pp. 109–119.

[11] RUGABER, S. Program comprehension. In *Encyclopedia of Computer Science and Technology*, M. Dekker, Ed., vol. 35 of *20*. Inc:New York, 1995, pp. 341–368.

[12] SAMSON, W., NEVILL, D., AND DUGARD, P. Predictive software metrics based on a formal specification. In *Information and Software Technology* (June 1987), vol. 29 of *5*, pp. 242–248.

[13] SPIVEY, J. *The Z Notation: A Reference Manual*, 2nd ed. Prentice Hall International, 1998.

[14] STOREY, M.-A., BEST, C., AND MICHAUD, J. Shrimp views: An interactive environment for exploring java programs. In *Ninth International Workshop on Program Comprehension* (Toronto, May 2001), IEEE, pp. 111–112.

[15] TILLEY, S. R. Domain-retargetable reverse engineering iii: Layered modeling. Tech. rep., Software Engineering Institute, Carnegie Mellon University, 1995.

[16] TIP, F. A Survey of Program Slicing Techniques. Tech. rep., CWI Netherlands, 1994.

[17] TUCKER, A. B. *The Computer Science and Engineering Handbook*. CRC Press, 1996.

[18] VINTER, R., LOOMES, M., AND KORNBROT, D. Applying software metrics to formal specifications: A cognitive approach. In *5th International Symposium on Software Metrics* (Bethesda, Maryland, 1998), IEEE Computer Society, pp. 216–223.

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