Demand-driven Specification Partitioning

Roland T. Mittermeir and Andreas Bollin

Institut für Informatik-Systeme
Universität Klagenfurt, Austria
{roland,andi}@ifi.uni-klu.ac.at

Abstract. The paper reflects on why formal methods are quite often not used in projects that better rely on their potential. The expressive density might not be the least among them. To allow users focussed reading, the concept of specification slicing and specification chunking is introduced. An initial evaluation shows that reduction in size obtainable varies, they can be marked with larger specifications though.

1 Introduction

The crucial role of requirements elicitation and documentation is well accepted [1, 2]. Some organizations even venture into requirements testing [3] to get them right. Specifying systems formally is less common though. While graphical notations like UML have reached a certain level of acceptance, specifications in languages expressing their semantics on the basis of a formal model are rare to find in industrial applications.

One has to be careful not to overgeneralize. There are certainly highly professional places around. On the other hand, there is lots of software written, where quick production of code [4] is more economical and possibly even more effective than bridging the magic leap [5] from requirements to design or from design to code by a carefully constructed formal specification. However, one gets worried when managers responsible for the production of dependable systems resort to complex explanations and make statements about compensating the disregard of formal specification by multiple (up to quintuple) modular redundancy, supported by solid reviews and heavy testing.

Certainly, many reasons can be voiced to justify avoiding formal methods. Arguments put forward are: we use N-version programming, we use only safe programming constructs, and we apply thorough testing. The fact that part of the software is written by domain experts lacking formal software-engineering education might also be among the reasons. It is rarely mentioned though.

Unfortunately, what is meant to be a valid excuse is built on shaky grounds. As safety critical systems are generally embedded systems, the reasoning of those managers (and their staff) is quite often dominated by arguments valid in classical engineering disciplines. There, triple (or higher) modular redundancy seems to be a safe bet if Poisson error rate in the individual components can be assumed. This assumption holds for most physical engineering artifacts as long as there are no common cause failures. Since faults in software are of conceptual

nature, one tries to solve this issue by having the individual components developed by different teams. However, as shown in [6], N-version programming is not sufficiently effective to get rid of the effect, common education principles have engraved on software developers.

Use of "safe programming strategies" is usually a circumscription for avoiding constructs involving pointers and hence dynamic storage management. This has certainly advantages by allowing to compute upper bounds for resource consumption. However, it lowers the language level used. Had developers a chance to reason about the correctness of structures close to the application domain as well as about the mapping of these structures via clichés to the low-level constructs they are avoiding, the resulting software might be less complex and hence less buggy in the end. Without such an intermediate level, limitations in the expressive power of languages may just widen the gap to be covered by some magic leap and thus leads to further errors.

Finally, heavy emphasis on testing certainly increases quality. However, without an adequately formalized specification, formal testing is bound to be confined to structural (white-box) testing. As is well known, white-box approaches and black-box approaches are complimentary [7]. However, without a formal specification, there is no sound basis to systematically drive black-box testing or to assess black box coverage.

Remains the educational argument. While this argument might hold true, it cannot be accepted as definitive excuse. People who learned how to program should be able to learn how to write a formal specification. Hence, there must be other reasons, why they shy away from "software-engineering mathematics" while faithfully accepting the equally complex mathematics describing the physics or mechanics of their application domain.

We postulate that there are motivational factors that might be rooted in the inherent linguistic complexity of formal specifications. The next section will look more thoroughly into this argument. Based on these considerations, we propose mechanisms to support specification comprehension. With proper tool support, these concepts allow software engineers, notably maintenance personnel, to query a complex specification and obtain only the part that is relevant to the specific question at hand. The arguments are demonstrated on a set of cases analyzed.

2 The inherent complexity of specifications

Let's start with an observation: people like to write code, but they do not like to read somebody else's code.

Though not based on a deep empirical survey, this statement rests on experience gained by talking to people and by observing students' as well as professionals' behavior during software maintenance. Why might this be the case? Again, based only on introspection, we postulate that it is easier to express ones owns concepts and ideas into the tight formality of a programming language than to reconstruct the concepts the original developer had in mind from the formal text written in low level code. This statement does not hold, if the orig-

inal programmer adhered strictly to some standard patterns or clichés. But it holds when the code expresses a concept previously unknown to the reader. It also applies when the concept is known, but the particular form used to express it is unknown to the reader. Bennett and Rajlichs evolutionary software development model [8] postulates that the transition from the evolution phase to the servicing phase takes place when the chief-architect leaves the team. This can be quoted as macroscopic evidence for our claim.

What arguments might be raised to explain this observation? The density of linguistic expressions has certainly an impact. Humans are used to listen and talk with equal ease in their natural language. Observing a human dialog, one notices that, irrespective of the specific grammatical constructs used, it boils down to a sequence of questions and answers or assertions and counter-assertions that finally converge to the core of the message. Thus, if both parties of a conversation are reasonably sure that the respective partners' frame of reference is sufficiently adjusted such that the concept to be conveyed has actually been transferred, they might pass on to the next topic [9].

With written communication, we do not have this chance of constant probing. However, the systemic functional theory of linguistics postulates that even at such low levels as in the sentence structure, we use a head phrase (the theme) which is further explained in the rest of the sentence (the rhematic part) [10, 11]. Thus, readers of different expertise can make use of the partial redundancy between theme and rheme, respectively of the incremental nature of information transfer happening in the rhematic part.

With programming languages, this redundancy providing reassurance that some partial comprehension is still on track towards full comprehension is missing. However, at least with small programs, the well defined execution sequence among statements allows for partial comprehension and thus for incremental growth of knowledge about this piece of software. Thus, we are well capable of obtaining some understanding by performing a desk-check in the form of a program walkthrough with some assumed values. In essence, this is to inspect a dynamic slice of the program. Inspecting the program without assuming specific values bound to the program's variables is much harder.

With specifications, we have no longer a built in clue for partial comprehension. Their compactness allows the writer to succinctly pin down an idea. Due to their declarative natures, the writer does not need to worry about order of execution. As the concepts to be specified are already in the specifiers mind, the most compact way to express them is most appropriate. For the reader of a specification, the assumption that a reasonably consistent picture is already in the readers mind does not hold. The specification either needs to be grasped in its entirety or some strategy has to be identified to structure it in a way supporting partial understanding.[1]

[1] The final exams of the specification class we are teaching contains a part where students have to write a specification and a part where students have to comprehend a given specification. Although, measured in length of text, these assignments differ

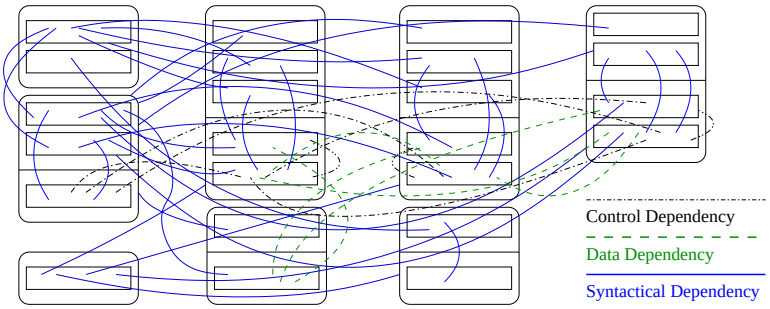

Fig. 1. Direct dependencies in the Z-specification of the birthday book [12].

Putting too much structure into a specification is usually understood to be a hint towards implementation. Hence, one has to be careful. To demonstrate our argument on a trivial example, one might consider the number of interrelationships between concepts used in the toy-specification of a Birthday-Book [12]. The BB might come close to what a minimal meaningful specification might be. The number of different dependencies between concepts (see Fig. 1) might be indicative of the fact, that it will be hard, to scale it up to an industrial size specification and still claim that the result is easy to comprehend.

Based on these considerations, we raise some arguments where partial comprehension of specifications are helpful. In the sequel, we present a concept to derive such partial specifications automatically from a full-fledged specification.

3 Demand-driven Specification Decomposition

Usually, specifications are considered to be an intermediary product of the early phases of software development. They serve to conquer and structure the problem at hand, to harness the developer's ideas, and to reason about closure properties of requirements and the correctness of a design. They come into play again during testing. But code-level patches render them quickly outdated. For software comprehension, one usually uses tools to analyze code. Why not raising the importance of specifications by granting them life throughout the full life cycle of the software product? Keeping them up-to-date will allow using them as maintenance road-map. Pursuing this aim, one certainly has to provide tools to support partial specification comprehension, tools similar to those supporting partial code comprehension.

Focussing on the maintenance stages, a specification might serve to identify hot spots for change. While automatic tracing of change propagation [13, 14] quickly reaches its limits, specifications are well suited to identify the boundaries of change propagations on the specification level and right into design, if the

by a factor of 10 to 20, the time allocated to them differs only by a factor of two to three. The correctness of the results differs not at all.

actual implementation is still faithful with respect to its specification. Likewise, if formal specifications are used in picking test suites, identifying the portion of the specification relevant with respect to a given changing requirement can drastically reduce the effort needed during regression testing. Pursuing these ideas, we are aiming to provide developers, notably maintenance personnel, with partial specifications that are

- substantially smaller than the full specification, and hence, easier to grasp,
- contain all relevant parts of interest,
- can be automatically derived from the full specification.

The informatics literature contains several concepts of partiality, aiming to provide an interested party just the perspective needed for a particular task. The notions of views, slices, and chunks immediately come to mind.

Views initially defined in the data-base area have been introduced to the specification literature by Daniel Jackson [15]. Data-base views are drawn from a given conceptual schema. As long as update operations via views are restricted, the view mechanism necessitates no additional integrity constraints on the schema. Jackson's specification views, though, are bottom-up constructions and the full specification of the system apparently has to allow for updates on the state space. Hence, developing software specifications via views requires additional support structure to maintain consistency of all views.

Slices have been introduced by Weiser [16] in the mid '80ies and obtained several extensions since. A slice yields the minimal part of a program that satisfies all requirements of a syntactically correct program with respect to a given slicing criterion. As such, it seems ideal for our current aim. The concept has been transferred to specifications in the work of Oda and Araki [17] and Chang and Richardson [18]. However, a direct transfer is problematic, since Weiser-slices depend on control-flow. This is not existent in specifications. Therefore, the definitions we could build upon are too liberal.

Chunks have been introduced by Burnstein [19]. Similar notions have been used in [20] for semantics preserving software restructuring. With code, a chunk comprises all those statements necessary to understand a semantically related portion of a program. The requirement of semantic completeness is weakened though. Again, with specifications the definitions need to be sharpened and recasted.

One might also mention *multi-dimensional hyperslices* [21] and the related concepts introduced in connection with aspect-oriented or subject-oriented programming. These approaches are less pertinent to our consideration though, since they are strictly generative. Our concern though is to ensure that the virtues of formal specifications can find a better use throughout the lifetime of a system.

Thus, none of these concepts of partiality developed in other sub-domains of informatics can be applied directly to specifications. However, they guide our considerations. In the sequel, the ideas leading to specification chunks and specification slices are presented. Readers interested in the formal definitions are referred to [22].

3.1 Components of Specifications

Formal specifications are expressions written in some formal language such that the primitives of the language have a formally defined semantics.

Apparently, each specification language has linguistic elements at lower granules than postulated in the previous sentence. We refer to them as *literals* of the language. Examples of literals are identifiers, linguistic operators, etc.

Minimal linguistic expressions that can be assigned meaning are called *prime objects* or *primes* for short. Primes are constructed from literals. They form the basic, semantics bearing entities of a specification. Examples are predicates or expressions. Particular specification languages will allow also for semantically richer primes. E.g. in Z, a schema or a generic type can be seen as a prime. These higher level primes are defined arrangements of literals and other primes. The important aspect is that prime objects are immutable as far as they constitute fundamental units (states and operations) specifications are built upon and that they constitute a consistent syntactic entity with defined semantics.

To support the comprehension process, independent prime objects, even if they might get complex, are usually insufficient. The peruser of a specification usually needs more than a given prime but less than the full-fledged specification to get an answer to the problem at hand. Usually, one needs a set of different primes which are to be found not necessarily in textually contiguous parts of the specification. We refer to such a set, falling short of being the full specification and falling short of any syntactical constraints observed between different primes as *specification fragment*.

Thus, the rest of the paper focusses on the following issue: given a certain (maintenance) question against a specification, what is the minimal set of primes to answer it. Following Weisers slicing idea, this fragment should have all properties of a syntactically correct specification, a property that is not fulfilled by the proposal of [18]. To solve this involves two further issues:

- What primes are needed with which kind of question?
- What is the appropriate data-structure for extracting semantically related primes and literals?

We postpone the first question, which boils down to whether one aims for slicing (i.e. wants to see the minimal portion of the specification affected by, say a changed requirement) or for chunking (i.e. looks for a minimal portion of the specification specifically related to a given part in the specification). This question becomes relevant only after a way is found to isolate an (arbitrary) part of interest and then complete this part in such a way that a minimal expression in the given specification language results that is syntactically correct and semantically complete with respect to the question at hand. The avenue towards this end is seen in the transformation of the specification into a graph and then a proper back-transformation to the original specification language.

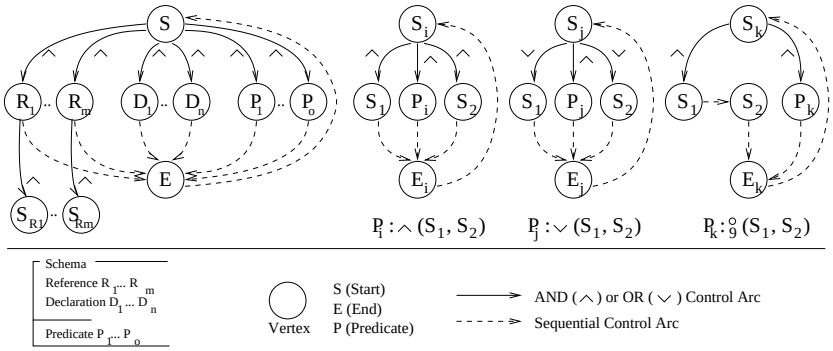

Fig. 2. Some transformations for Z-specifications into a Specification Relationship Net.

3.2 Augmented Specification Relationship Nets

Solving the question: given some prime, which set of primes is needed to provide a maintainer with sufficient information to understand just that part of the system, requires to explore this primes neighborhood. It seems appropriate to transform the specification in a graph, analyze the dependencies in this graph, and perform a back-transformation of the resulting graph-fragment into the original specification language.

Using an AST as intermediate representation seems to be an obvious choice. However, the dominant structuring principle of ASTs is, according to the microstructure of programming languages, control. This is inappropriate for declarative specifications. Good layout will help comprehending specifications, but, as with program indentation, layout has in general no semantic bearing. Getting it wrong, though, would distort readability drastically. Hence, partitioning has to retain the overall layout of the specification.On the other hand, there is a certain degree of freeness considering the placement of compound specification elements. There is no implicit canonic order comparable to the order implied by control in imperative programs.

For these reasons, we did not venture into extending ASTs to suit our purpose, but defined a graph, the *Augmented Specification Relationship Nets* or *ASRN*. An *ASRN* captures all information contained in a specification but separates the semantics contained in primes from their textual appearance in the linear or two-dimensional representation of printed text. The textual information contained in a specification is captured in an *SRN*, the *Specification Relationship Net*. There, primes are represented by arc-classified bipartite graphs such that every prime is a miniature lattice with a unique syntactic start-node and a unique syntactic end-node. Between start- and end-node are the nodes (or subgraphs) representing the elements out of which this prime is constructed. These might be literals or, with higher-order primes, references to composed primes. Depending on the relationship between the entries of the respective prime, the links from start-node to sub-prime are *AND*- or *OR*-nodes. Fig. 2 schematically shows how

a schema definition, consisting of several included schemata ($S_{R1}...S_{Rm}$), declarations ($D_1...D_n$), and predicates ($P_1...P_o$) would be represented in the *SRN*. It also shows *SRN*s for schema-conjunction, disjunction and composition.

The *ASRN* is an *SRN* where several semantic dependencies contained in the original specification are made explicit. These are *type* definitions, *channel* definitions (lead to the declarations of all variables in the scope of a vertex v), *value/def* definitions (definitions relevant at that vertex), and *use* arcs (leading to all vertices used at vertex v). Thus, while we assume that the specification to be transformed is correct, the *SRN* does not yet allow for static checks. Augmenting the *SRN* by the links mentioned, semantic layers are introduced that allow to identify in the *ASRN* all those vertices from which a given prime (vertex) is type-dependent, data-dependent, or simply syntactical dependent[2]. This seems to be sufficient to address the question of chunking. What is still missing, though, is a notion representing the equivalent of control dependence inherent in code-slicing.

To address this issue, one has first to reflect on what constitutes the equivalence of control in specifications. Apparently, preconditions of operations define, whether an operation will be executed. How the precondition is represented and how it will be evaluated in the final program is immaterial at this point. But for the specification, it suffices to observe that a situation not satisfying a precondition implies that the transformation (or action) expressed by this operation should never be executed while a situation satisfying the precondition is a situation where the operation might be executed, i.e., it potentially obtains control. In certain specification languages, e.g. *VDM*, pre- and post-conditions are explicitly defined. In others, like *Z*, they have to be computed. As the algorithm must not depend on the specifiers adherence to style conventions, we opted for the heuristic that predicates involving no state changes or external activities are considered pre-conditions. Those involving state changes (i.e., contain decorated variables) are considered post-conditions.

3.3 Chunks and Slices

With these preparatory remarks, chunks and slices can be defined. Both, chunks and slices, are abstractions of the given full specification defined "around" a specific point of interest. In program slicing, this is referred to as slicing criterion. It refers to a variable in a given statement (at position). Here, we look at a variable (literal, l) within a given prime (vertex v_l or *articulation prime*). The maintainer might be interested in the interaction of l with some other variables of the specification. Usually, those will be other variables referred to in the same prime, but those can be any variables appearing anywhere in the specification. One might summarize them under the term "potentially interesting targets". They can be listed in an additional argument Θ in the *abstraction criterion* (*slicing-, resp. chunking-criterion*).

[2] This encompasses dependencies from vertices needed for the proper definition of the prime under consideration. Thus, it covers type-dependencies as well as channel-dependencies.

Definition 1. *An abstraction criterion for a specification is a triple* (l, Θ, Γ)*, where* l *is a literal in a prime of the specification (represented in the ASRN by vertex* $v_l \in V$*).* Θ *is a subset of the set of symbols (variables) defined or used in the ASRN, and* Γ *is a set of dependencies,* $\Gamma \subseteq \{C, D, S\}$*, with* C *denoting control dependency,* D *denoting data dependency, and* S *denoting syntactical dependency.*

If in the above definition the literals and primes listed in Θ all occur in the prime, l is taken from and Γ is set to a value different from the full set $\{C, D, S\}$ we refer to it as chunking criterion, one obtains a conceptual (hyper-)plane along the dependency categories listed in Γ, originating at the articulation-prime and transitively extending over the full specification. This hyper-plane is referred to as *static specification chunk* or *SChunk*. Compared with the notion of Burnstein-Chunks, where locality was aimed at, this provides for full projection.

The following definition captures a chunking concept similar to Burnstein's ideas on the specification level. It, the *BChunk*, is made up of the intermediate context of a prime according to some filter criterion.

Definition 2. *A static BChunk referred to as* $BChunk(PL, \Theta, \Gamma)$ *of* S *on chunking criterion* (PL, Θ, Γ) *(where* $v_p \in V$ *and* $V_P \subseteq V$ *and* $\Gamma \neq \varnothing$*) is a subset of vertices* $V_{PL} \subseteq V$ *such that for all* $v \in V$ *holds: vertex* $v \in V_{PL}$ *iff* v *is either directly data-dependent with respect to the variables defined in* Θ *on* V_P *and* $D \in \Gamma$*, or directly control-dependent on* V_P *and* $C \in \Gamma$*, or syntactical dependent on* V_P *and* $S \in \Gamma$*.*

The definition of BChunks just focusses on which other vertices than the criterion need to be included in V_l. This suffices, since the selected vertices represent primes in the original specification and these primes are related by the *SRN* covering the whole specification. The external representation of the specification chunk is the result of a back-transformation of all those vertices (primes) in the *SRN* that belong to V_l. The *SRN*-arcs will assure that the result will be a particularly focussed specification fragment.

As the above definition for chunks was chosen to transitively cover the full specification according to the respective chunking criterion in a particular (set of) dimension(s) of interest, it can be directly extended to the definition of a specification slice. As with program slices, we require that a specification slice is a syntactically correct specification which is, with respect to the slicing criterion, also semantically complete. This will be obtained, if all kinds of dependencies are included in the specification criterion, i.e., $\Gamma = \{C, D, S\}$:

Definition 3. *The* static specification slice $SSlice(l, \Theta)$ *on a slicing criterion* (l, Θ) *corresponds to the full static specification chunk* $SChunk(l, \Theta, \{D, C, S\})$*.*

As slices adhere rigidly to the constraint of syntactical correctness, the resulting specification stays syntactically correct and remains (with respect to the specification criterion) understandable. Nevertheless, as all dependency types are included in the slice, the resulting slice is normally bigger than the chunks derivable. With tightly interwoven specifications, the slice might get as big as the original specification.

4 Case Studies

Based on the definitions mentioned above, a toolkit for slicing and chunking Z-specifications has been developed. We applied the partitioning algorithms on several specifications, among them are the Birthday-Book [12], the Elevator [12, 18] and the Petrol-Station, a specification we used as course assignment.

None of these specifications comes close to industrial size. They just served as test- and demonstration cases during method- and tool development. Nevertheless, they can be considered as proof of concept for the arguments raised and to show that the concepts put forward do reduce complexity of size and, therefore, the overall complexity of the specification comprehension task, if full understanding is not really needed.

The literature in the field of software metrics is vast. The various measures of size are at least correlated. The proposals to measure inherent complexity are quite different though. However, there is agreement that size is a prominent scaling factor even for other forms of complexity [23].

Measuring size of a specification by number of characters, number of lines (as analog to *LOC*) or number of predicates (as analog to *KDSI*) seems not very attractive. Such measures miss the crucial point of compactness of formal specifications referred to in the introduction. Hence, we decided to measure the complexity of a specification by the number of concepts and the number of interrelationships between concepts needed to formally trace it in order to obtain a meaningful specification fragment. Chunks or slices would qualify for such meaningful fragments. Hence, the number of vertices and arcs in an *ASRN* seems to be a fair candidate metric.

However, one should not take their raw value, since many arcs in the *ASRN* are just needed to capture layout information and external structure of the specification. Their use is not burdening the comprehender. The links that implicitly relate concepts contained in the specification, be they implicit data- or control-dependencies, or be they syntactical dependencies that get, when interpreted, a particular semantic meaning, cause interpretative load.

Table 1 shows the number of vertices / number of nodes in the *ASRN* for the three specifications mentioned, and also quotes the number of control-, data- and syntactical dependencies.

In the *ASRN* representation, even the simple BB specification consists of 84 vertices and 302 arcs. Since the BB is about the simplest non-trivial specification one might think about, these numbers seem already high. But one must not be frightened. Only 142 arcs have semantic significance. The remaining 160 capture layout information needed for the back-transformation. Thus, only 142, i.e., 128 for syntactical dependencies, 7 for control and 7 for data-dependence carry intellectual load. Due to this different significance of link-weights, the reduction-percentage is of limited significance. Readers might get a better appreciation of the benefit of this approach when comparing the overall reduction in the various link categories tabulated in the second line of each experiment.

When applying a static Slice $SSlice(n, birthday)$ (where l represents the prime object "$birthday' = birthday \cup \{name? \mapsto date?\}$" in the Add-schema), the

Original Spec.	BirthdayBook	Elevator	Petrol-Station
ASRN Metric	84V / 302A 128s - 7c - 7d	344V / 2835A 634s - 386c - 1160d	134V / 668A 251s - 36c - 120d
$SSlice(l, \theta_1)$	58V / 201A 87s - 5s - 4d	240V / 2105A 464s - 260c - 881d	109V / 542A 211s - 26c - 92d
$SSlice(l, \theta_2)$	53V / 184A 79s - 4c - 4d	175V / 1407A 330s - 77c - 565d	103V / 503A 198s - 16c- 82d
$SChunk(l, \theta_2, \{S, C\})$	38V / 130A 57s - 2c - 2d	35V / 146A 52s - 5c - 9d	39V / 182A 71s - 5c - 26d
$SChunk(l, \theta_2, \{S, D\})$	46V / 158A 62s - 4c - 2d	113V / 698A 174s - 63c - 90d	86V / 403A 154s - 16c - 43d

V ... vertices (including syntactical comment node vertices) - A ... arcs in the *ASRN*
l ... vertex in the *ASRN* representing a prime object in the specification
θ_1, θ_2 ... slicing with different set of literals (see text for explanation)
s, c, d ... number of arcs representing syntactical, control and data dependencies.

Table 1. Complexity of the *ASRN* representation of three different specifications. The rows represent the number of nodes and arcs after applying slicing and chunking functions on the specification.

overall complexity of the net (and the specification) is distinctly reduced. (More important is the reduction of the overall number of arcs representing syntactical-, control- and data-dependency). The second slice on birthday is also on this predicate in the Add-schema, but $\Theta 2$ consists only of $\{birthday\}$ in contrast to $\Theta 1 = \{name, birthday\}$ in the first slice. The same holds for the application of two chunking functions onto the same abstraction criterion. When transforming the sliced *ASRN* back to Z, the remaining specification only consists of the BB state-schema, the InitBB schema, the Add- and the Remove schema. The schemata Find and Success are pruned from the specification and the remaining specification has been reduced by about 30 percent. The difference in Θ has with this example only a minor effect.

The difference between $\Theta 1$ and $\Theta 2$ is significant though in the much larger Elevator specification[3].

With the full panel of variables, reduction is not impressive. Slicing for a single variable yields a reduction of 50 % though. The slices have been computed with the prime $Requests' = Requests$ in the FloorButtonEvent-Schema. If, for this schema, only syntactical semantic and control dependencies are considered, the resulting chunk has only 35 (instead of 344) vertices and 146 (instead of 2835) links. The Petrol Station assumes a medium position in this table.

Reductions of even higher magnitude can been found when looking at the numbers of dependencies in the *ASRN*. When looking at the first chunk of the elevator example, one will find out that 376 control dependencies and 1160 data de-

[3] In our example we used $\Theta 1 = \{Requests, CurrentFloor, Door, UpCalls, Down\text{-}Calls\}$ and $\Theta 2 = \{Requests\}$ for the generation of the slices.

pendencies are reduced to just 5 control-dependencies and 9 data-dependencies. In that case this is a reduction of at least 98% – an impressive number of dependencies whose elimination is tremendously reducing the complexity of the remaining specification.

Apparently, the reduction factor depends on various aspects. The compactness of a specification and the interrelationship between state variables not being the least among them. Nevertheless, our preliminary analysis suggests that the effect obtainable by slicing and chunking rises with the size of specifications under consideration. Thus, our claim that these concepts are meant for "industrial size" specifications seems substantiated.

5 Summary-Conclusion

The paper introduced a mechanism for automatically deriving semantically meaningful fragments of formal specifications. The work was motivated by helping software developers to comprehend large specifications in situations where their current interest is confined to a particular aspect only. Nevertheless, these developers have to be sure that no relevant portion of the specification which directly or indirectly influences the part they are focussing on, will be ignored in their partial comprehension.

Chunks and slices have been discussed as prototypical abstractions that proved their value already in program understanding. Initial assessments are encouraging, since the effects of focussing attention tends to increase with larger specifications. With large specifications, the *ASRN* can no longer be explicitly displayed. However, it can still be computed efficiently, since the approach builds only on proven compiler technology.

Departing from the archetypical specification fragments of chunks and slices, we plan to further refine the approach presented in this paper, such that more specific issues arising during software maintenance can be answered by even narrower focussed specification fragments containing all the information needed for the task at hand.

References

1. Boehm, B.W.: Software Engineering Economics. Prentice Hall, Englewood Clifss, NJ (1981)
2. Kotonya, G., Sommerville, I.: Requirements Engineering. 2nd edn. John Wiley & Sons, Ltd. (1998)
3. Graham, D.: Requirements and Testing: Seven Missing-Link Myth. IEEE Software **19** (2002) 15 – 17
4. Boehm, B.: Get Ready for Agile Methods, with Care. IEEE Computer **35** (2002) 64 – 69
5. Potter, B., Sinclair, J., Till, D.: An Introduction to Formal Specification and Z. Prentice-Hall Intl. (1991)

6. Knight, J.C., Leveson, N.G.: An Experimental Evaluation of the Assumption of Indepencence in Multiversion Programming. IEEE Trans. on Software Engineering **SE-12** (1986)
7. Beizer, B.: Black-Box Testing: Techniques for Functional Testing of Software Systems. John Wiley & Sons, Inc. (1995)
8. Bennett, K.H., Rajlich, V.T.: Software maintenance and evolution: A roadmap. In Finkelstein, A., ed.: The Future of Software Engineering 2000. ACM press (2000) 73 – 87
9. Stewart, C.J., Cash, W.B.: Interviewing: Principles and Practices. 2nd edn. Wm. C. Brown, Iowa (1978)
10. Daneš, F.: Functional sentence perspective and the organization of the text. In Danes, F., ed.: Papers on Functional Sentence Perspective, Academia, Publishing House of The Czechoslovak Academy of Sciences, Prague (1970) 106–128
11. Halliday, M.: An Introduction to Functional Grammar. Edward Arnold (1985)
12. Spivey, J.: The Z Notation: A Reference Manual. Second edn. Prentice Hall International (1992)
13. Pirker, H.: Specification Based Software Maintenance: A Motivation for Service Channels. PhD thesis, Universität Klagenfurt (2001)
14. Pirker, H., Mittermeir, R.T.: Internal service channels: Principles and limits. In: Proceedings International Workshop on the Principles of Software Evolution (IWPSE'98), IEEE-CS Press (1998) 63 – 67
15. Jackson, D.: Structuring Z Specifications with Views. ACM Trans. on Software Engineering and Methodology **4** (1995)
16. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, IEEE (1982) 439–449
17. Oda, T., Araki, K.: Specification slicing in a formal methods software development. In: 17th Annual International Computer Software and Applications Conference. IEEE Computer Socienty Press (1993) 313–319
18. Chang, J., Richardson, D.J.: Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California (1994)
19. Burnstein, I., Roberson, K., Saner, F., Mirza, A., 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, IEEE press (1997)
20. Mittermeir, R., Rauner-Reithmayer, D.: Applying concepts of soft-computing to software re(verse)-engineering. In: Migration Strategies for Legacy Systems. TUV-1841-97-06 (1997)
21. Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N degrees of separation: Multi-dimensional separation of concerns. In: Proc. 22^{nd} Internat. Conference on Software Engineering, ACM and IEEE press (1999) 107 – 119
22. Bollin, A., Mittermeir, R.T.: Specification Fragments with Defined Semantics to Support SW-Evolution. In: ACCIT/IEEE Proc. of the Arab-International Conference on Computer Systems and Applications (AICCSA'03). (2003)
23. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigourous & Practical Approach. Second edn. International Thompson Publishing Company (1996)