JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
*J. Softw. Maint. Evol.: Res. Pract.* 2008; **20**:77–105
Published online 11 February 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.363

**Research**

# Concept location in formal specifications

Andreas Bollin*, †

*Alpen-Adria Universität Klagenfurt, Universitätsstrasse 65-67, 9020 Klagenfurt, Austria*

**SUMMARY**

**When kept up-to-date, formal specifications can act as valid artifacts for maintenance tasks. However, their linguistic density and size impede comprehension, reuse, and change activities. Techniques such as specification slicing and chunking help in reducing the number of relevant lines of text to be considered, but they expect the point of change to be known *a priori*. This contribution presents a process model for concept location within formal *Z*-specifications. It also considers those situations when the location is not even roughly known. The identification is comparable to the identification of regions with high cohesion. The approach is based on the idea of first transforming the specification to an augmented graph and, secondly, on the generation of spacial clusters. Copyright © 2008 John Wiley & Sons, Ltd.**

*Received 4 January 2007; Revised 5 November 2007; Accepted 5 November 2007*

KEY WORDS: formal specifications; concept location; slicing; chunking; cluster generation

## 1. INTRODUCTION

The use of formal methods helps in raising the overall quality of the software and the related software development process. Despite several myths and problems [1,2], formal specifications have been used successfully more than once—they were and are used as test-driver generators and form the basis for verification and validation steps [3,4]. What is often ignored, though, is the fact that formal specifications provide valid resources during development and maintenance phases [5,6].

One of the benefits of formal specifications is their density in writing down thoughts as they allow the authors to express requirements in a keen and compact way. However, this property—in combination with the raising size of today's systems—is one of the reasons for considering formal specifications to be too complex. Tools and approaches are needed to deal with complexity more than ever.

*Correspondence to: Andreas Bollin, Institut für Informatik-Systeme, Alpen-Adria Universität Klagenfurt, Universitätsstrasse 65-67, 9020 Klagenfurt, Austria.
†E-mail: Andreas.Bollin@uni-klu.ac.at

Formal specifications are complex 'entities'. Even simple textbook specifications (of about 20–200 lines of specification text) typically contain dozens of dependencies between the related specification elements. Within the scope of this work, these elements are called *prime objects* and they are, like program statements, the basic construction elements [6].

In order to guarantee the benefits of formal specifications (during evolutionary steps taking place throughout the life cycle), it gets necessary to evolve and also to *change* the specification. This change requires to first comprehend the specification. This can be demanding, and it gets really difficult when the maintenance personnel is not the author of the specification.

There are several ways of dealing with the compactness of specifications. When the location of the specification text to be changed is already known, smaller parts of the specification can be detached by slicing or chunking [7–10]. Specification slices and chunks are well-defined types of abstraction, and for *Z*-specifications [11] they do reduce the size of the text by 20% up to 60% [12].

However, typically, a change request might not contain a reference to the lines of specification text to be altered. Requests are often formulated in terms of changing/adding/deleting features or concepts [13]. For that reason concept and feature location play an increasing role in software maintenance, and it will also play an important role during specification maintenance activities.

This contribution introduces an approach facilitating the comprehension process of complex formal specifications. Research dealing with specification slices and chunks so far assumes that either the point of change is known *a priori*, or that this point is to be found in an intuitive way (by linearly scanning through the specification text). The main contribution of this work consists of two closely related parts: firstly, a process model for specification comprehension, and secondly, fit into it, an approach for specification concept identification aiming at exactly those cases when not even a surmise can be made about the concepts' locations.

The contribution is structured as follows: Section 2 summarizes the idea behind concept location and defines the term 'formal specification concept'. Section 3 introduces a process model for concept location within formal specifications. It also introduces the notion of a specification cluster, a subset of specification elements, which will be treated as a concept candidate. Section 4 presents a clustering algorithm for *Z*-specifications. Section 5 demonstrates the applicability of the approach and addresses the issue of scalability. Section 6 concludes the work and provides hints to further variations of the approach.

## 2. CONCEPTS AND CONCEPT LOCATION

The Webster's dictionary defines the notion of a *concept* as an *abstract or generic idea generalized from particular instances*. In that sense it is a logical entity conceived in the mind. When talking about concepts in software engineering, the term *feature* is used very often. A feature then represents a selectable concept, which means that it is up to the user (and thus up to specific control sequences) whether the feature is executed or not. According to Kozaczynski *et al*. [14] *concept location* typically denotes a process that maps these domain concepts (or patterns) to code positions. The process of identifying patterns or abstract concepts 'behind' the code is sometimes also called *plan recognition*.

By following the constructivist's views, existing approaches [13,15–17] do build hierarchies of concepts that are related to parts of a unified representation of the program (in most cases the AST).



DOI: 10.1002/smr

By aggregating base concepts, higher-level concepts are defined and stored in a library. When searching for concepts, AI techniques and fuzzy reasoning can be applied [18]. Code positions are typically part of several concepts, and to speed up the combinatorial problem when searching for concepts, special heuristics or constraint-satisfaction problem solving algorithms can be used [19].

For the purpose of this paper, the notion of a *concept* also follows the constructivists' views. Concepts do get unique concept names and are, according to a taxonomy, organized in hierarchies of primitive concepts $C$ (e.g. $C_{Bubble-Sort}$ and $C_{Quicksort}$ belong to $C_{Sorting-Algorithm}$, and $C_{Sorting-Algorithm}$ belongs to $C_{Algorithm}$). Individuals are instances of a concept $C$. By making use of the description logic defined in [20, p. 19], the following general definition of a concept can be provided (the domain, one is reasoning about, is expressed by the symbol $\Delta$):

*Definition 1*. A concept $C$ is an intensional description of a class of individuals. It consists of a set of primitive concepts $p$ and the transitive closure of their antecessor concepts. $C^I$, the interpretation $I$ of the concept $C$, consists of the domain $\Delta^I$ and $p^I$, the interpretation of the primitives $p \in C$.

According to Kozaczynski *et al*. [14, p. 1068], concepts 'are not tightly related by syntactic structures, they are more closely connected by semantic relationships such as control flow, data flow, and calling relations'. These relationships, sometimes also called *constraints*, facilitate the process of concept location.

### 2.1. Identification of concepts

The technique of concept location is, in most cases, an intuitive process. Experienced users hardly can explain how concepts are identified; they navigate through the code and quickly separate relevant parts from irrelevant ones. When experience and familiarity with the code is not sufficient, more 'structured' approaches are needed. Following [13], these approaches can be classified as follows:

(a) *Pattern matching*: Here, string-pattern matching algorithms are used (e.g. 'grep'-ing for relevant keywords/identifiers) in order to identify concepts of pre-assumed names.
(b) *Dynamic analysis*: They are based on looking at the dynamic behavior of the code. The program is executed, and the program trace is analyzed. Here, a popular method is called software reconnaissance [21]. The code is annotated and executed twice, the first time without the feature and the second time with the feature executed. Those parts of the code that are only executed in the second run are good candidates for starting the search for the assumed concept.
(c) *Static analysis*: Here, typically control and/or data flow is considered. By starting at the main entrance point of the program, tracing through the code, and by following explicit and implicit dependencies, it is assumed to speed up the identification of the exact location of the concept.

When taking a closer look at the above-mentioned approaches, it turns out that not all of them can be applied to formal specifications:

(a) String-pattern matching works well for concept location within formal specifications. However, there are drawbacks. Without hints to suitable homonyms and synonyms this



DOI: 10.1002/smr

technique fails. In addition to that, specifications are very compact. This compactness is achieved by making use of mathematical symbols, abbreviations, and powerful operations. Developers of formal specifications typically do not label them with long and self-explanatory names, which also impedes looking for string patterns.

(b) Dynamic analysis requires execution of the code. In general, specifications are not executable. Owing to their declarative nature (as there is no explicit ordering of statements), a paper-and-pencil run is also not possible. Dynamic analysis is impeded.

(c) The static analysis approach is mainly based on the identification of two dependencies that are not predominant in declarative systems: control and data-flow dependencies. However, as will be explained in Section 4.2, these dependencies can be re-constructed calculating of pre- and post-conditions within operations, and thus they can then be used to generate different types of specification abstractions [6].

By making use of pattern matching and slicing or chunking, the identified parts very likely represent concepts or features. However, there are two challenges concerning these static techniques:

- The number of dependencies within formal specifications is typically overwhelming. There is no heuristic telling the peruser which of the dependencies to follow.
- Slicing and chunking is based on a slicing/chunking criterion, thus on a specific position in the specification text. In most cases, this position is not known *a priori*.

As will be explained in Section 4.2, the static information yielded by a formal specifications is sufficient to solve these problems.

In most cases it is easier to state that some piece of text represents a concept than reasoning the other way round. With formal specifications this means that a specification slice might represent a concept, but it is unreasonable to state that every concept can be covered by a specific slice. Thus, for the scope of this contribution, the approach in fact leads to *concept candidates*. The final validation of these candidates is, as is the same with program concept location, up to the user.

### 2.2. Concepts in formal specifications

First, let us start with a definition of a specification concept that borrows from the very general definition of a concept at the beginning of Section 2.

*Definition 2*. A *formal specification concept* is a coherent, abstract (or generic) pattern of specification text that is generalized from particular instances of the specification. It can be understood and recognized as a whole even when standing alone.

The definition is closely related to those of a Burnstein chunk [22] or a specification cliché [9, p. 50]. The first part of the definition deals with its representational form, the second part ensures that a specification concept has a meaning that is unambiguously understood within its context.

With respect to Definition 1 it should be clear that the interpretation of a specification concept strongly depends on the interpretation of its related parts. These parts or basic objects are called primes of a specification.



DOI: 10.1002/smr

*Definition 3*. A *specification prime* (also called prime object) represents the basic entity of a specification. It is built out of literals of the specification and forms logical, syntactic, or semantic units.

In that sense a specification prime constitutes those primitives that form the smallest, basic concepts expressed in a formal specification. It is a *syntactically coherent sequence of literals within a state-based specification, forming semantic entities that can be paraphrased by a short sentence in natural language*. An example would be a $Z$-predicate checking whether some names are in a specific set (called *known*) or not: '$name? \notin known$'.

When primes are aggregated, they do form the so-called *higher-level primes*. An example for a higher-level prime is the schema expression '$[\Xi Elevator | Request \cup UpCalls \cup DownCalls = \emptyset]$'. It consists of two basic primes, $\Xi Elevator$ and a predicate describing properties of some sets. A more detailed introduction to $Z$-primes can be found in [9, p. 43].

With such basic elements a more precise definition of a specification concept can be provided.

*Definition 4*. A *formal specification concept* $C_{FS}$ within a formal specification $\Psi$ is a concept according to Definition 1, where the domain $\Delta$ is the domain of the formal specification $\Psi$, and the primitive concepts $p$ are represented by prime objects.

*Definition 5*. The *interpretation* of the formal specification concept $C_{FS}$ is determined by aggregating the primes' interpretations according to Definition 1 ($C^I_{FS} == (\Delta^I, \{p : Prime | p \in C_{FS} \cdot p^I\})$).

In order to be able to reject primes as concept candidates, the following definition is yielded:

*Definition 6*. Every specification prime $p_i : \Psi$ is element of $C_{FS}$ when at least one of the following conditions holds:

(i) There are (syntactic and/or semantic) dependencies between prime $p_i$ and primes already in $C_{FS}$.
(ii) The absence of prime $p_i$'s semantics in $C_{FS}$ impedes the recognition of the concept in mind.

Thus, the semantics of the concept or concept candidate is constructed by the subsumption of the semantic interpretations of its related components.

### 2.3. Concept location within formal specifications

Formal specification concept location is still cumbersome. A program-like static technique for searching for concepts is impeded by the huge number of dependencies and the often missing 'starting point'. However, this hurdle can also be seen as an advantage. As noted in [14], concepts often express themselves by their constraints, their semantic relationships. And looking for specific groups of constraints can be compared with clustering the specification.

*Definition 7*. A *cluster* is a group of the same or similar objects gathered or occurring closely together.

In our case, the objects are primes. Similarity and closeness might be expressed by several properties, for example, similar type or number of dependencies the objects depend upon. For the scope of this paper the reachability via dependencies will be used as an argument for closeness.



DOI: 10.1002/smr

*Definition 8*. A *specification cluster* is a group of primes. A prime $p$ is element of the cluster when the number of (direct or indirect) dependencies to elements already in the cluster is higher than the number of dependencies to elements outside the cluster.

Generally speaking, the clusters we are looking for are sets of primes with high cohesion with respect to direct and indirect relationships.

Specification clusters (as well as slices and chunks) are being built by the aggregation of prime objects and by regarding specific types of dependencies in between them. This property coincides with the definition of a specification concept. This is the main reason for stating that *specification clusters* (*as well as slices and chunks*) *are to be treated as concept candidates*.

The remaining arguments (reinforcing that concept location can be supported by looking for specification clusters) are as follows:

- Several dependencies can be considered once at a time as it is not always sensible to follow only a single dependency.
- By clustering, related objects are brought together, and dissimilar objects are separated. No artificial ordering of prime objects is introduced.
- Clusters do not necessarily have strict boundaries. The same holds for concepts and their related primes—they can also be part of other concepts.

There are several clustering algorithms, each one with its own advantages and disadvantages (for an introduction, see [23]). Cluster algorithms depend on at least two inputs: a pattern vector and the number of clusters to be identified. In our case this allows for adjusting the granularity of the search space—something that will be useful when narrowing down the search space. The clustering approach presented in this contribution is based on clustering spatial properties of a directed graph. The approach is embedded into a process model for specification concept location, and the next section introduces the underlying model in more detail.

## 3. CONCEPT LOCATION PROCESS MODEL

The search for relevant pieces of specification text is never purely top-down or bottom-up, it is a mixture of both—and consists of iterative steps. In addition to that, and being confronted with an unknown specification, one has to start with an initial concept in mind, a concept that is validated against concept candidates. Figure 1 presents a model, which consists of four main activities (and which is a refinement of the model already presented in [24]).

- *Domain-level request*: The feature or concept to be located in the specification text is formulated, and typically comes from ‘outside’.
- *Concept formation*: The peruser has to form a mental image of the problem, thus it requires to comprehend the request from the outside world. With this, the process of concept location is about to start.
- *Concept location*: It is an iterative process consisting of several sub-activities:
  - *Concept candidate evaluation*: With the problem in mind the peruser takes the concept candidate (at the beginning the intuitively most suitable specification fragment) and checks



DOI: 10.1002/smr

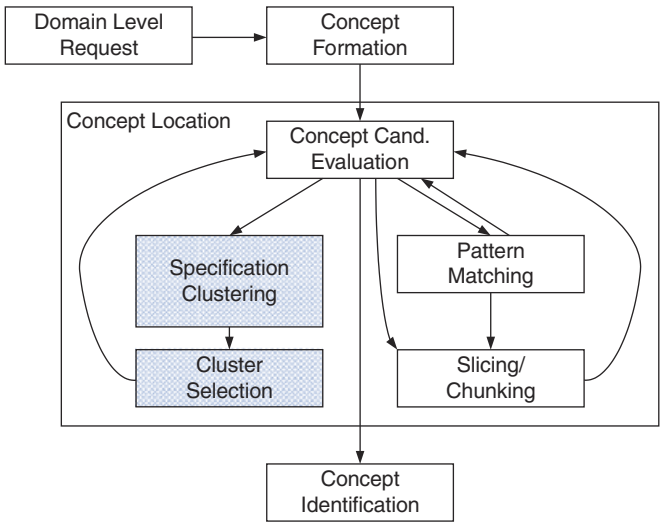


Figure 1. Concept location as an iterative process. Concept candidates are identified and evaluated with respect to their suitability. Either clustering techniques or patter-matching and slicing/chunking are applied. (The approach described in details in Section 4.3 is marked gray, slicing/chunking is described thoroughly in [9].)

whether it matches to the concept in mind or not. Then he or she decides between accepting the candidate or stepping (again) into the concept location process.
  - *Specification clustering*: The candidate either contains no parts of interest or it is still too large. Clustering is applied in order to narrow down (or broaden) the search space.
  - *Cluster selection*: Clustering yields a set of clusters. Each of these clusters has different semantics the peruser has to understand. One or more suitable clusters have to be selected as potential candidates, others dismissed.
  - *Pattern matching*: When the search space is large, pattern matching strategies can be applied. The peruser browses or 'greps' through the candidate(s) and looks for relevant labels or identifiers. This might help separating suitable from less suitable parts of the document.
  - *Slicing/chunking*: When possibly relevant locations in the specification text are identified, slicing and chunking can be applied. This helps in modifying the search space again, but also guarantees (when necessary) syntactical completeness and correctness.

- *Concept identification*: The location process results in a set of candidates that very likely do contain those parts (primes) to be changed. These parts have to be marked (or labelled) as being part of the change process.

The steps of activities are introduced in a top-down manner, but it is also possible to start bottom-up. The peruser then has to decide whether he or she is satisfied with the candidate or not. Clustering and slicing/chunking techniques can, as a next step, be used to enlarge or narrow down the set of relevant concept candidates. In any case, one is following the iterative steps of the model.

As the clustering approach is not to be confused with the strategy of 'divide and conquer', it needs more explanation. The data set is not divided into several separate parts that are not overlapping.



DOI: 10.1002/smr

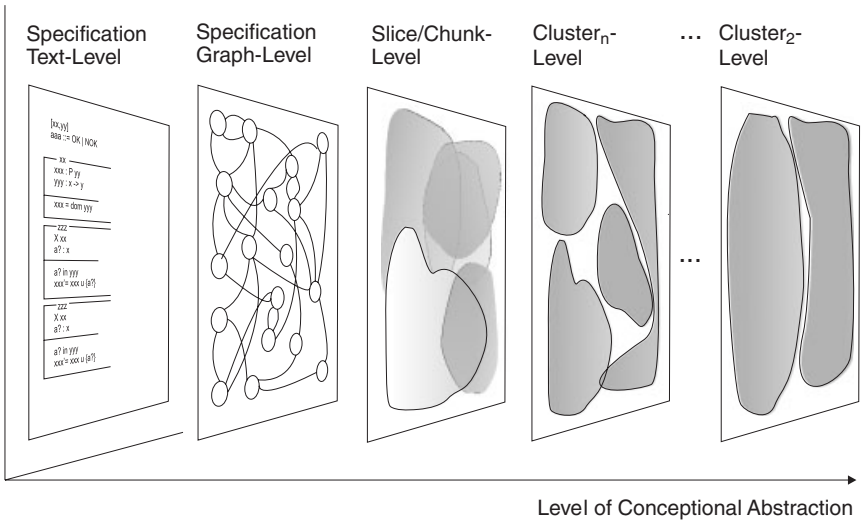


Figure 2. Formal specifications can be transformed to a graph mapping primes to vertices. These graphs form the basis for dependency analysis. It is possible to carve out slices, chunks, or clusters by aggregating vertices.

In the above process model a partitioning also takes place (which is the reason for speeding up the search), but the partitions might change and might contain parts of candidates that were excluded one loop before.

The basis for the specification comprehension process is the specification itself (specification text or graph level in Figure 2). The representation can be either seen as (linear) specification text, or as a graph containing all relevant information. According to Mittermeir and Bollin [6], a graph is more likely the 'true' representation of a specification, as there is no linear ordering. However, as will be explained in Section 4.1, both representations can be transformed loss-lessly to each other. When starting to look for abstractions with well-known semantics, slices and chunks (slice/chunk level in Figure 2) are considered. The bad news is that slices and chunks are typically quite large. Experiments indicate that a $Z$-chunk tends to contain 50 prime objects, and slices are often of the same size as the original specification [5]. Additionally, matching the general description of a concept of the world to a slicing criterion (the exact location of at least one prime object) is quite difficult.

Clusters add an orthogonal view onto this problem (cluster levels in Figure 2). The peruser decides about the granularity and might start with $n$ clusters, which also have a semantic driven by the included prime objects. As we will see later, there is no 'optimal' number of clusters. However, it is still up to the user to decide whether some clusters match the target concept or not. He or she might start to increase the number of clusters and thus 'sharpen' the concept candidates. On the other hand, he or she might reduce the number of clusters, increasing the level of the abstraction. The disadvantage of providing the number of clusters at the beginning rapidly gets an advantage as with this input the level of abstraction can be 'controlled' within some boundaries.



DOI: 10.1002/smr

Finally, when identifying a suitable set of cluster candidates, the target prime objects are tied down. Then the level of abstraction can be decreased again, and, with the primes in mind, slices and chunks can be generated in order to identify those parts that are affected by a potential change.

## 4. CLUSTERING ALGORITHMS

With these ideas in mind the approach has been evaluated on formal *Z*-specifications to be found in textbooks and the literature [8,25]. For clustering (besides the number of clusters), the pattern vector is the most relevant input. It is based on properties of the graph, and the subsequent section introduces the necessary background. Section 5 finally addresses the issues of applicability and scalability of the approach.

### 4.1. Basic transformation

At first the formal specification is transformed into a representation called *specification-relationship net* (*SRN*). The *SRN* is a directed graph that captures all primes present in the specification. It is defined independently of particular specification languages; however, the rule set to translate a specification in a syntax- and semantic-preserving manner to an *SRN* has to be defined in a language-dependent manner. The *SRN* is then further augmented by *semantical structural information*, yielding an *augmented SRN* (*ASRN*). This information explicates dependencies between prime objects due to *type* and *variable declarations*, as well as definition and use (*def*/*use*) *information*.

One major motivation for the generation of an *ASRN* was to deal with a representation that expresses, on the one hand, the absence of a specific ordering within specification elements itself and, on the other hand, the close relationships between specification primes. In fact, both forms of representation (text and ASRN) are interchangeable. The transformation basically maps primes to vertices in the graph. Concerning the structure of the net, the idea behind an *SRN* is that primes are enclosed by typed vertices (*Start* and *End* vertices), thus defining some sort of higher-level primes. Figure 3 shows how the *SRN* is used to represent a schema of a *Z*-specification. *Z*-constructs are translated to *SRN* primitives. Primes that consist of other primes (such as *Z*-schemata) contain vertices that reference to those primes. Thus, the general structure ensures that a hierarchy of primes and its enclosing structures are being built.

To summarize, an *ASRN* is an extension of an *SRN*, where vertices are getting attributes describing the 'usage' of identifiers belonging to that vertices. A typical augmentation is demonstrated in Figure 4. Besides the line numbers ($L$), it consists of the source text (as annotations $A$), the definitions of variables ($D$), the use of variables ($U$), type declarations ($T$), and input/output channel definitions ($C$).

### 4.2. Dependencies

As reachability in the *ASRN* is put on a par with the notion of relatedness (which will be later an argument for cluster generation) this section explains when and under which conditions vertices in the graph are connected by arcs. In fact, in an *ASRN* there are four classes of arcs, each of them representing different types of relationships.



DOI: 10.1002/smr

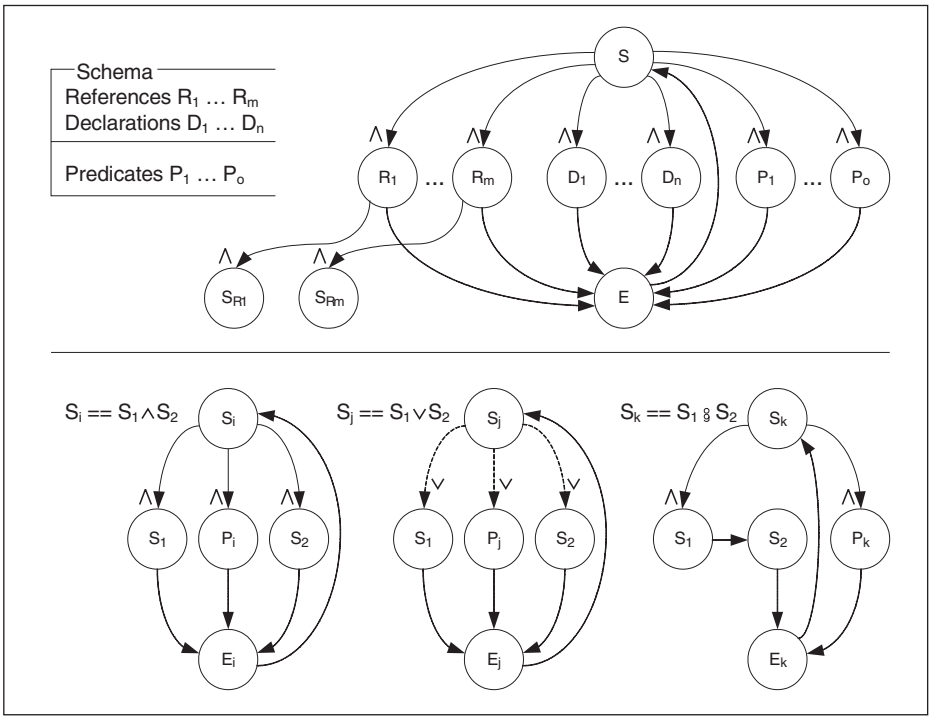


Figure 3. Mapping *Z*-specifications (at top left) to an SRN. The idea is to enclose the specifications' prime objects between structure-preserving vertices. Logical combinations (at bottom) are resolved by typed arcs.

### *4.2.1. Sequential control arcs*

Although the ordering of predicates within specifications is not relevant, pieces of text still rely on a logical or linear order that has to be preserved (e.g. the start of a schema box has to be before its predicates). This is done by making use of *sequential control arcs*. In order to deal with logical (boolean) operations, the sequential control arcs can be typed as either AND or OR control arcs. All arcs in Figure 3 do belong to this class.

### *4.2.2. Declarational arcs*

Identifiers typically have to be declared before they can be used. Depending on the specification language, there will always be dependencies that exist only due to the syntactic and semantic rules of the language (including the notion of scope). This does not refer to deep semantics of a specification, but it defines those dependencies that a good *Z*-checker would identify. From the parsers' perspectives, however, these dependencies are beyond simple syntax. It is legitime to entitle this dependency as of being semantically structural. They are expressed by (*semantically*) *declarational arcs* in the *ASRN*. In Figure 4 there are several arcs (labelled with *SD*) expressing this



DOI: 10.1002/smr

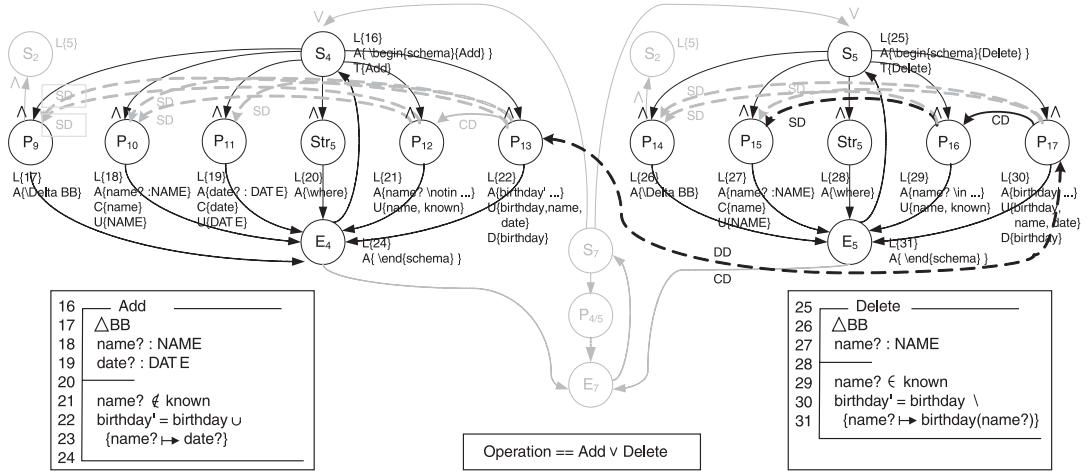


Figure 4. The *Add* and *Delete* operation schemata (bottom) of the birthday-book specification (see Appendix B) are mapped to an *ASRN* (top). Vertices do get annotations, and arcs are labelled according to the class they belong to. Dependencies are made explicit. (For reasons of readability irrelevant parts are grayed out.)

type of dependency. For example, prime $P_{16}$ uses the identifier *name*?, which has to be declared first (to be seen at prime $P_{15}$).

### 4.2.3. *Control-dependency arcs*

The definition of *control dependency* can be kept rather simple when based on pre- and post-condition analysis between primes. By following the arguments in [8], in $Z$ a prime $q$ is control dependent on a prime $p$ when $p$ potentially decides whether $q$ is applied or not. This means that post-condition primes (after-state primes) are control dependent on pre-condition primes. The same idea holds when elements of the specification are logically combined. If, again in $Z$, a schema object $S_1$ is conjuncted to a schema object $S_2$, all post-conditions of $S_1$ are control dependent on the pre-condition of $S_1$ and $S_2$, and all post-conditions of $S_2$ are control dependent on the pre-condition of $S_2$ and $S_1$.

The identification is straightforward and, due to the structure of the graph, reduced to a reachability problem. One has to look for primes that do have defined (annotated by a $D$) identifiers (thus they are post-condition primes) and for primes that do not refer to any after-states. When they are reachable according to their scope then there is a control dependency between them. In Figure 4 control dependency is expressed by the arcs labelled with *CD*. E.g. $P_{17}$ is control dependent on $P_{16}$, as it is reachable from $P_{16}$ via sequential control arcs and $P_{16}$ does not define any identifier, indicating a pre-condition prime.

### 4.2.4. *Data-dependency arcs*

The identification of *data dependency* is quite similar to the identification of control dependency. A prime $q$ is data dependent on a prime $p$, when data potentially propagate from $p$ to $q$ through a



DOI: 10.1002/smr

series of state changes. Again the *ASRN* eases the identification, as one has to look for identifiers at primes that are, within declarational boundaries, redefined at one prime (thus annotated by a $D$) and used at another prime (annotated by an $U$). Figure 4 shows an *ASRN* where the two operation schemata *Add* and *Delete* are logically combined. Besides control dependencies, there is data dependency between primes $P_{17}$ and $P_{13}$, as the identifier *birthday* is re-defined ($D$) and used ($U$) at the opposite primes each.

With the structure of the *ASRN* in mind one can also start to look for sets of primes that are closely (in terms of distance in the graph) related to each other. Relatedness then has to do with spatial reachability across different types of arcs, and the subsequent section explains how clustering techniques can be used to identify these regions.

### 4.3. Cluster identification

Clustering is an approach that groups pattern vectors that do have similar characteristics in some respects. According to Halkidi *et al*. [23], the process of cluster identification is divided into several steps:

- *Feature selection*: Here, all attributes on which the clustering is to be performed have to be defined. As the relevant information (for the problem at hand) has to be encoded this is a critical step.
- *Cluster algorithm selection and clustering*: There are several types of clustering algorithms. Examples are partitioning clustering (e.g. *KMeans*), fuzzy clustering (e.g. *FCM*), or hierarchical clustering. A proximity measure and a clustering criterion have to be selected.
- *Validation*: The resulting clusters are not known *a priori*. Hence, the final partitions have to be evaluated with respect to usability and appropriateness.
- *Interpretation*: The resulting clusters do have specific semantics that have to be interpreted correctly. One has to answer the important question: What does this cluster mean?

#### *4.3.1. Feature selection*

The first and very sensitive step is called feature selection. The arguments provided in Section 2 indicate that one is interested in grouping those parts of the specification that are closely correlated via dependencies. In other words, for the *ASRN* it is reachability that counts. It makes sense to take prime objects (vertices in the *ASRN*) as data points and examine *SRN* paths in the graph. For every prime, the cost for reaching other primes in the graph can be calculated by identifying the length of the shortest path. For every vertex this yields a distance vector, and each vector discriminates the prime from the others. Figure 5 presents a small graph (which is typical for structures to be found in *ASRN*s) and its related pattern vectors.

By searching for the shortest path, a simple cost function can be defined. The definition presumes that *minlength* is a function determining the shortest path between two vertices in the *ASRN*.

*Definition 9*. For all $n$ vertices $v$, with $v$ element of the *ASRN*, an $n$-dimensional pattern vector $x_v$ ($v=1\ldots n$) is defined. For every vertex $x_v$ the set of paths $P$ to vertices $x_l$ ($l=1\ldots n$) is calculated, and $x_v[l]$ is defined to be the minimum of the lengths of the paths in $P$ ($x_v[l]=$ *minlength*(*paths*$(x_v, x_l)$)). The length is set to $-1$ when there is no path.



DOI: 10.1002/smr

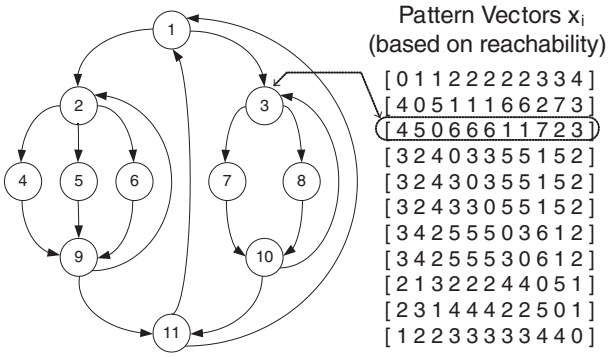


Figure 5. Pattern vectors for a simple *SRN* graph. The minimal length of the path between two vertices is taken as the attribute discriminating the relationship between the two vertices, stored in the pattern vectors $x_i$.

For every vertex in the graph, the vector represents the minimal costs in reaching all other vertices. As an example, in Figure 5 the vertex number 3 gets the feature vector $x_3$ (line 3) attached. The vector tells us that vertex 1 is reachable from vertex 3 by the cost of 4 (over vertices 7, 10, and 11), and vertex 11 is reachable from vertex 3 by the cost of 3.

### *4.3.2. Cluster algorithm and criterion*

The second step is cluster algorithm selection. As one is interested in separating the data with respect to the pattern vectors, we have to deal with partitioning algorithms. As it is very likely that primes can be part of different concepts (with different probabilities) at a time, the fuzzy $c$-means algorithm [26] has been chosen in order to express memberships of different extents.

For reasons of simplicity, the proximity measure is the Euclidean distance. The clustering criterion is defined so as to minimize the following objective function $J_m$:

$$J_m = \sum_{i=1}^{N} \sum_{j=1}^{C} u_{ij}^m \| x_i - c_j \| \tag{1}$$

where $x_i$ represents the pattern vector defined above. $m$ is a real number greater than 1 (in our case set to 2.0), $u_{ij}$ is the degree of membership of pattern vector $x_i$ in cluster $j$, $c_j$ is the center of cluster $j$, $N$ is the number of pattern vectors, and $C$ is the number of cluster centers. Fuzzy clustering is an iterative optimization, where the values for $u_{ij}$ and $c_j$ are calculated.

Concerning the example in Figure 5 one can apply the *FCM* algorithm with the argument of three clusters to be identified. Figure 6 summarizes the results of the *FCM* algorithm. It yields the cluster centers and, for every vertex, a membership value $u_{ij}$. Those indices containing values equal to the maximum value of the rows $n$ of matrix $U$ are candidates for the cluster $n$. In our case, this leads to three clusters containing the following vertices:

- Cluster 1: $V_3, V_7, V_8, V_{10}$.
- Cluster 2: $V_1, V_{11}$.
- Cluster 3: $V_2, V_4, V_5, V_6, V_9$.



DOI: 10.1002/smr

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Centers c:** | | | | | | | | | | |
| 3.09 | 4.08 | 1.30 | 5.08 | 5.08 | 5.08 | 1.47 | 1.47 | 6.07 | 1.15 | 2.11 |
| 0.83 | 1.60 | 1.69 | 2.56 | 2.56 | 2.56 | 2.67 | 2.67 | 3.37 | 3.47 | 1.94 |
| 3.00 | 1.43 | 3.99 | 1.81 | 1.81 | 1.81 | 4.99 | 4.99 | 1.07 | 5.37 | 2.03 |
| | | | | | | | | | | |
| **Membership Matrix U:** | | | | | | | | | | |
| 0.10 | 0.06 | 0.86 | 0.05 | 0.05 | 0.05 | 0.87 | 0.87 | 0.05 | 0.67 | 0.10 |
| 0.78 | 0.15 | 0.09 | 0.15 | 0.15 | 0.15 | 0.09 | 0.09 | 0.22 | 0.26 | 0.78 |
| 0.13 | 0.80 | 0.05 | 0.81 | 0.81 | 0.81 | 0.04 | 0.04 | 0.74 | 0.08 | 0.12 |
| | | | | | | | | | | |
| **Maximum of U:** | | | | | | | | | | |
| 0.78 | 0.80 | 0.86 | 0.81 | 0.81 | 0.81 | 0.87 | 0.87 | 0.74 | 0.67 | 0.78 |

Figure 6. Results of the calculation of the pattern vector's centers for the *SRN* (by applying $FCM(data, 3)$ to the data set) in Figure 5.

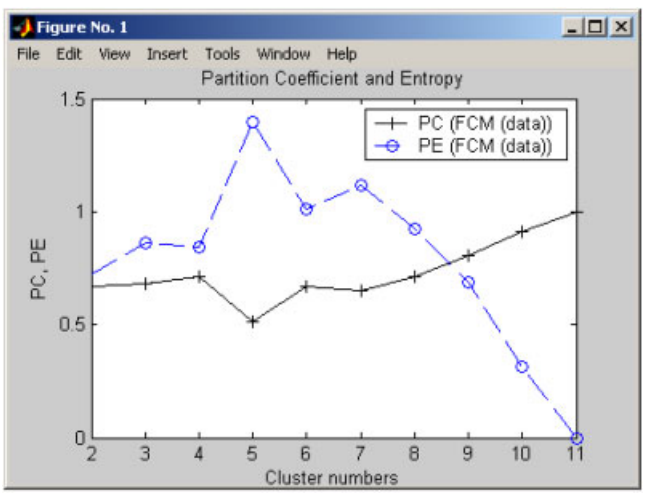


Figure 7. PC and PE are measures describing the fuzziness of the generated clusters. For the graph in Figure 5, the values are calculated.

This is exactly what was expected. In our case a user of the graph can easily observe that the vertices in the clusters are reachable by just a few arcs. However, with larger graphs this gets much more difficult, and $a(n)$ (semi-)automated approach is likely to be helpful.

### *4.3.3. Cluster validation*

The objective of clustering is to seek clusters where the related data vectors exhibit a high degree of membership. However, when there are $n$ vertices, it is possible to apply $FCM(data, C)$ with argument $C$ running from $2 \ldots n$. The important question is then how 'good' the generated clusters are. Here, the answer of the appropriateness of clusters cannot be given automatically, but there are clues indicating special properties of the cluster. These properties describe, e.g. how 'fuzzy' a cluster is, or how 'crisp'.



DOI: 10.1002/smr

The approach presented so far does not suggest a specific number of clusters to be initially generated. In fact, the question of the 'ideal number of clusters' is interesting, but hard to predict. There are two possibilities to deal with this situation:

- Tell the user, that $2 \ldots C$ clusters are possible and let him or her decide freely about the cluster's generation.
- Calculate all possible sets of clusters and provide some hints about the clusters' internal properties. Then the user can decide about the cluster's generation.

The latter strategy seems to be more appropriate, as it still lets the decision up to the user and provides as much information as possible at that time. Typically, properties are described by the so-called *validity indices*. Two helpful indices are the *partition coefficient PC* and the *partition entropy PE*. These are defined as follows [23, p. 137]:

$$PC = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} u_{ij}^2 \tag{2}$$

$$PE = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} u_{ij} * ld(u_{ij}) \tag{3}$$

The *PC* index (2) tells about how fuzzy the cluster is. The closer the value to $1/C$, the crisper is the clustering. As can be seen in Figure 7, the index is between 0.5 and 1. The higher the number of clusters, the sharper the result. Theoretically, *PC* is a monotone increasing function when all clusters are fuzzy. In our case *PC* has a 'sharp' bend at 5, indicating that there the fuzzyness decreases. It might be a good idea to start looking around that number of clusters first.

The PE index (3) works the other way round: the nearer the values of *PE* to the upper bound $ld(C)$, the bigger is the absence of any clustering structure. Here a number of 2–4 clusters seems to be most interesting.

#### *4.3.4. Cluster interpretation*

Finally, the generated clusters have to be interpreted. The pattern vectors (that are grouped into clusters) tell us about the similarity in costs of reaching neighboring vertices. In our case this means that vertices with similar distance vectors are closer to each other. The semantics of the cluster is then defined by (i) the semantics of the primes (thus the ASRN's vertices) and (ii) the type of dependencies in between them. Hence, it is the same as with specification slices and chunks. Clusters do get their meaning in a neat way: by aggregating the meaning of the subgraph of the *ASRN*.

## 5. EVALUATION

### 5.1. Case study

The *Elevator* specification is a formal specification of a simple elevator control system to be found in textbooks and has been taken to discuss (quasi-) dynamic specification slicing in [8]. The task is to detect a bug as the elevator is sometimes moving into the wrong direction when pressing a button. Hence, all we know is that a *button* is pressed and that the *direction* of the *movement* is not as expected. The task of looking for the reason(s) of this malfunction is difficult, especially



DOI: 10.1002/smr

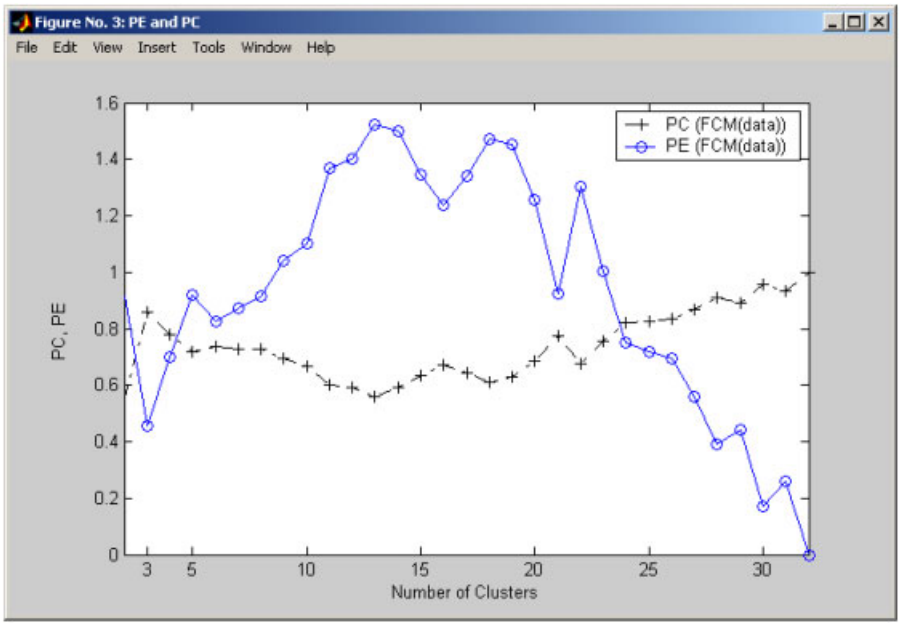


Figure 8. *PC* and *PE* values for 32 different clusters of the *Elevator* specification—with the focus on start vertices.

when one is not the author of the specification. In order to identify the location of the bug, one has to understand the concepts behind the specification. The concept location model presented in Section 3 is taken for their reconstruction. An annotated *Elevator* specification (including the hint to the bug) is provided in Appendix C.

Supported by a prototype tool (for details, see [9, p. 171]), the following activities are handled:

1. Understand the general concepts and the structure behind the specification.
2. Locate concept candidates and match them to the bugs description.
3. Narrow down the location of the concept.
4. Identify all relevant primes.
5. Change the relevant parts of the specification.

The process model does not help in avoiding to read the specifications' text, but it helps in narrowing down the search space quickly. Looking at the ASRN one realizes that the specification is quite complex. It contains 349 vertices, 1096 control, and 1212 data dependencies (direct and indirect). 144 vertices represent predicate primes, 32 vertices are start vertices.

At first, for reasons of orientation, the general structure behind the specification has to be understood. One decides to look at the syntactic structure of the specification and to carry out a simple clustering regarding the 32 start vertices (as they represent higher-level primes). The goal is to identify those regions that are (spatially) closely related. When taking a look at Figure 8, one can see that the clusters are rather crisp when generating three of them (also see Figure C1 in Appendix C).



DOI: 10.1002/smr

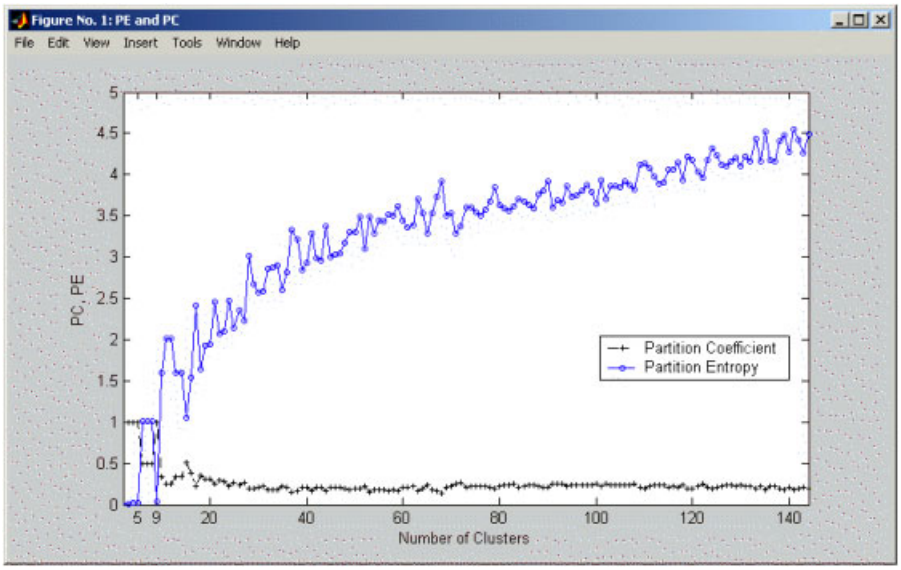


Figure 9. By looking at the *ASRN* of the *Elevator* specification, 144 different clusters (with the focus on primes representing predicates) can be generated and their PC and PE values calculated.

There are three closely related regions, and mapping the vertices back to the specification results in the following three *concept candidates* (containing the set of high-level primes):

1. Cluster 1 contains the primes: *BasicMoveUp*, *BasicMoveDown*, *ChangeUpToDown*, *ChangeDownToUp*, *RestartMovingUp*, *RestartMovingDown*, and the primes related to their logical combination.
2. Cluster 2 contains the primes: *Elevator*, *InitElevator*, *ElevatorButtonEvent*, *FloorButtonEvent*, the related given-set declarations, and their logical combination.
3. Cluster 3 contains the primes: *NoRequestOrCalls*, *OpenDoor*, *CloseDoor*, and their logical combination.

The first one deals with the movement of the elevator, the second one with the states, and the third one deals with opening and closing doors. As there is no problem with the doors, the primes in concept candidate 3 are excluded from the search in the future. This is one of the benefits of the definition of a specification concept: it is often easier to state that a set of primes is NOT within the concept to match.

None of the candidates matches the original concept, thus we step further into the cycle and generate clusters regarding all 144 predicate primes and their dependencies to other primes. Figure 9 shows the values for the crispness of the generated clusters and we notice that 5 or 9 clusters seem to be interesting. We can now adjust the granularity. We generated five clusters and regarded those identifiers with data-dependency arcs:

- *Cluster 1*: Deals with door states (*Dir*, *Door*, contains 20 primes—excluded).
- *Cluster 2*: Deals with the elevator's movement (*UpCalls*, *DownCalls*, *CurrentFloor*, 26 primes).



DOI: 10.1002/smr

- *Cluster 3*: Deals with the events (*Requests*, *CurrentFloor*, 11 primes).
- *Cluster 4*: Coordinates the idle state (*CurrentFloor*, 23 primes).
- *Cluster 5*: Coordinates the doors and events (*CurrentFloor*, *Dir*, *Door*, 64 primes—excluded).

From this the identifiers *CurrentFloor*, *Dir*, *DownCalls*, and *UpCalls* of clusters 2 and 3 seem to be most relevant with respect to the concept of our problem in mind—however, still too many primes are to be regarded. We decide to produce clusters that are smaller ($n = 9$). This leads to nine clusters (and again focus on identifiers with data dependencies):

- *Cluster 1*: Moves the elevator down (*CurrentFloor*, *DownCalls*, 12 primes—excluded).
- *Cluster 2*: Operates on the door (*CurrentFloor*, *Dir*, *Door*, 64 primes—already excluded).
- *Cluster 3*: Operates on the door (*Door*, 9 primes—already excluded).
- *Cluster 4*: Starting to move (*UpCalls*, *DownCalls*, *CurrentFloor*, 14 primes—*candidate*?).
- *Cluster 5*: Idle state (*CurrentFloor*, 10 primes—excluded).
- *Cluster 6*: Idle state and initialization (*CurrentFloor*, 13 primes—excluded).
- *Cluster 7*: Request handling (*Requests*, *CurrentFloor*, 11 primes—*candidate*?).
- *Cluster 8*: Operates on the door (*CurrentFloor*, *Dir*, *Door*, 64 primes—already excluded).
- *Cluster 9*: Changing the direction (*Dir*, 11 primes—excluded).

We evaluate the clusters: when the elevator is already moving then there are no problems. Using pattern matching and the findings of above, clusters 1 up to 3, and 5, 6, 8, and 9 might be excluded from the search. Hence, we suppose that the problem is located within cluster 4 or 7. Cluster 4 contains 14 primes and we take a closer look at it—thus reading the related specification text.

Finally, we identify one prime (in Figure C1 is the vertex entitled '343: L(55/39) FloorButton-Event' at the left) that sets the direction of the movement within the floor button event operation schema. With that we identified potential primitives of a concept candidate. There is still no guarantee that the candidate matches the problem description, and thus we decide to generate a Burnstein chunk on this prime element. It yields four schemata: *Elevator*, *InitElevator*, *FloorButtonEvent*, *and CloseDoor*, and thus reduces the size of the specification drastically.

By skimming through the remaining specification we find out that one predicate in *FloorButton-Event* has the bug inside. Owing to a copy–paste error, the reaction to an event was wrong. Instead of handling *upCalls* correctly, *upCalls* were treated as *downCalls*.

The process of concept location aims at speeding up the identification of relevant parts of the specification. However, there is the chance of digging into parts that are not relevant, too. Clusters provide some feedback and can give some hints about their semantic interpretation. But still it is up to the user to match the concept carved out of the specification text to the concept in mind.

### 5.2. Reflection

The usefulness of the suggested model depends, at least, on the following two aspects: firstly, the scalability of the approach, and secondly, the scalability of the algorithms behind. Really big formal specifications are rare, but the approach has been tested with several specifications of growing sizes. Among the largest is the specification of the ICT window manager of the 'Andrew distributed system' developed at the Carnegie-Mellon University. Its ASRN contains more than 500 primes (213 are higher-level primes) and 2633 dependencies. The specification is described in [27, p. 183ff].



DOI: 10.1002/smr

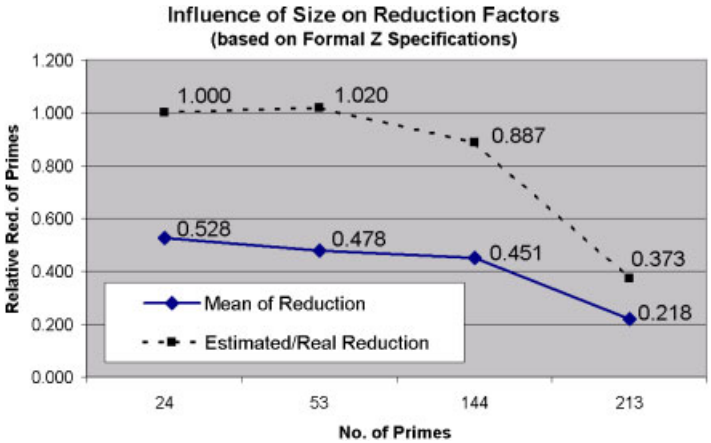


Figure 10. The effect of reduction increases with the size of the specification. The bigger the specification, the lower the $Mean\ of\ Reduction = Primes_{after}/Primes_{before}$. When the reduction increases at the same ratio than the specification, the values for the estimated reduction would be 1. Values less than 1 indicate better performance.

The question of scalability is directly related to the existence of conceptual structures behind the underlying specifications. Clustering, slicing, and chunking algorithms are based on the assumption that there are interconnected regions within the specification. Thus, the approach scales when (a) these regions exist, and (b) when, with growing sizes of the specification, these regions stay at manageable size.

(a) In [12], it has already been shown that chunks typically consist of 50 primes in the mean. Although it is possible to construct specifications that do contain regions of more than several dozens of interwoven primes, none of the case studies analyzed in [12] and looked at so far contained even complexer constructs. Developers tend to structure their solutions—which confirms the basic assumption that these regions might represent their concepts in mind. In well-designed specifications they have to exist, otherwise any structuring technique will fail.

(b) A simple experiment can show the attainable reduction with specifications of raising sizes. For the specifications (including the ICT window manager), all possible regions (considering different combinations of dependencies) can be generated. Figure 10 demonstrates the achieved reductions. The mean decreases from 0.528 (for a simple specifications with 24 primes) to 0.218 for the ICT window manager specification (with 213 primes). In addition to that it also shows that the reduction works better with the growing sizes. One might expect that, when the size of a specification doubles, the estimated sizes of the resulting regions might double. It is even better, and Figure 10 demonstrates this by comparing the factor of the estimated sizes to the real ones. For the ICT window manager, the reduction is 2.68 ($=\frac{1}{0.373}$) times better than expected.

The reduction factor depends on several aspects. The compactness of a specification and the inter-relationship between primes not being the least among them. Nevertheless, the analysis suggests that the effect obtainable rises with the size of specifications under consideration. The regions stay at manageable size.



DOI: 10.1002/smr

What remains is the questions concerning the time complexity of the underlying algorithms. In fact, the methods in the background are efficient enough to deal with typically available formal specifications. The fuzzy clustering approach has a run-time complexity of $O(n)$, and, as we are pre-calculating all $C=(n-1)$ clusters, we get a run-time complexity of $O(n^2)$. The pattern vector calculation is the crucial and time-consuming part. However, the slicing/chunking framework [5] is based on reachability arguments and thus already computes the necessary values in $O(n^2)$. For the *Elevator* specification it took about 100 s on a Pentium 4, 3.2 GHz, 1 GB RAM, to perform the ASRN transformation, cluster generation, and indices calculation—still faster than reading the whole specification.

## 6. CONCLUSION

This paper introduces the notion of a formal specification concept and presents a process model for concept location. The basic idea is to transform the specification into a directed graph (where prime objects are represented by vertices), and thus facilitate the generation of partial specifications (clusters, slices/chunks). Concepts are identified by looking for slices, chunks, and clusters, which are defined in such a way that those primes are regarded that are related to other primes with respect to specific types or numbers of dependencies.

The approach of specification concept identification is yet at the beginning and the next steps will be as follows: firstly, much more specifications will have to be examined to prove the usability of the model. Secondly, the pattern vector is crucial and at the moment all dependencies have the same weight. Adjusting the clusters by using weighting might make the approach more flexible. Finally, an empirical study will have to be conducted in order to prove its applicability by users other than the author of the workbench.

Specifications are complex buildings, but they are no lost cause. It is the author's belief that, by using the most suitable approaches available, several myths mentioned in the introduction will have to be re-written.

## APPENDIX A: BIRTHDAY BOOK

The birthday book (*BB* for short) out of [11] 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*

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



DOI: 10.1002/smr

At the beginning the database is empty.

*InitBB*

$$BB$$

$$known = \varnothing$$

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

*Add*

$$\Delta BB$$
$$name? : NAME$$
$$date? : DATE$$

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

*Delete*

$$\Delta BB$$
$$name? : NAME$$

$$name? \in known$$
$$birthday' = birthday \setminus \{name? \mapsto birthday(name?)\}$$

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

*Success*

$$result! : REPORT$$

$$result! = OK$$

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

$$FunctioningDB == ((Add \wedge Success) \vee (Delete \wedge (Success))^*$$

## APPENDIX B: ELEVATOR SPECIFICATION

The elevator specification [8] describes a simple system consisting of one elevator. The elevator reacts to button events corresponding to calls for the elevator. The call for the elevator is made on a floor. Additionally, a request to stop at a specific floor can be made inside the elevator.

There are 10 floors in the system. The elevator can move up or down and its door can be open or closed.



DOI: 10.1002/smr

$$MaxFloor == 10$$
$$Direction ::= up \mid down$$
$$DoorState ::= open \mid closed$$

The state space consists of the current floor, a set of pending requests, pending up- and down-calls, the current state of the movement and the door.

*Elevator*

$$CurrentFloor : \mathbb{N}_1$$
$$Requests : \mathbb{P}\,\mathbb{N}_1$$
$$UpCalls : \mathbb{P}\,\mathbb{N}_1$$
$$DownCalls : \mathbb{P}\,\mathbb{N}_1$$
$$Dir : Direction$$
$$Door : DoorState$$

---

$$CurrentFloor \leq MaxFloor$$
$$max\ Requests \leq MaxFloor$$
$$max\ UpCalls \leq MaxFloor$$
$$max\ DownCalls \leq MaxFloor$$

Initially, the elevator is at the first floor, and the door is open. There are no pending requests and calls.

*InitElevator*

$$Elevator$$

---

$$CurrentFloor = 1$$
$$Requests = \varnothing$$
$$UpCalls = \varnothing$$
$$DownCalls = \varnothing$$
$$Dir = up$$
$$Door = open$$

There are two passenger events that are to be handled. Firstly, a passenger might request to stop the elevator at a specific floor from inside the elevator (*ElevatorButtonEvent*). Secondly, a passenger calls for the elevator (*FloorButtonEvent*).

*ElevatorButtonEvent*

$$\Delta Elevator$$
$$Floor? : \mathbb{N}_1$$

---

$$Floor? \leq MaxFloor$$
$$CurrentFloor' = CurrentFloor$$
$$Requests' = Requests \cup \{Floor?\}$$
$$UpCalls' = UpCalls$$
$$DownCalls' = DownCalls$$
$$(Dir' = Dir) \land (Door' = Door)$$



DOI: 10.1002/smr

*FloorButtonEvent*

$$
\begin{array}{l}
\Delta Elevator \\
Floor? : \mathbb{N}_1 \\
CallDir? : Direction \\
\hline
Floor? \leq MaxFloor \\
CurrentFloor' = CurrentFloor \\
(CallDir? = down) \Rightarrow \\
\quad (UpCalls' = UpCalls) \wedge \\
\quad (DownCalls' = DownCalls \cup \{Floor?\}) \\
(CallDir? = up) \Rightarrow \\
\quad (UpCalls' = UpCalls) \wedge \\
\quad (DownCalls' = DownCalls \cup \{Floor?\}) \\
Requests' = Requests \\
(Dir' = Dir) \wedge (Door' = Door)
\end{array}
$$

$$
\begin{array}{l}
PassengerEvent == ElevatorButtonEvent \\
\qquad \vee FloorButtonEvent
\end{array}
$$

**In the above operation schema *FloorButtonEvent* the bug is hidden. Instead of**

$$
\begin{array}{l}
(CallDir? = up) \Rightarrow \\
\quad (DownCalls' = DownCalls) \wedge \\
\quad (UpCalls' = UpCalls \cup \{Floor?\})
\end{array}
$$

**the specification contains the following (buggy) lines:**

$$
\begin{array}{l}
(CallDir? = up) \Rightarrow \\
\quad (UpCalls' = UpCalls) \wedge \\
\quad (DownCalls' = DownCalls \cup \{Floor?\})
\end{array}
$$

The movement of the elevator is complex. An elevator might move up or down (*BasicMoveUp* or *BasicMoveDown*) when it is servicing requests or calls.

*BasicMoveUp*

$$
\begin{array}{l}
\Delta Elevator \\
\hline
Dir = up \\
Requests \cup UpCalls \neq \varnothing \\
CurrentFloor \leq max\,(Requests \\
\qquad \cup UpCalls) \\
CurrentFloor' = min\,\{x : \mathbb{N}_1 \mid x \in \\
\qquad (Requests \cup UpCalls) \\
\qquad \wedge\, x > CurrentFloor\} \\
Requests' = Requests \setminus \\
\qquad \{CurrentFloor'\} \\
UpCalls' = UpCalls \\
DownCalls' = DownCalls \setminus \\
\qquad \{CurrentFloor'\} \\
(Dir' = up) \wedge (Door' = Door)
\end{array}
$$



DOI: 10.1002/smr

*BasicMoveDown*

$$
\begin{array}{l}
\Delta Elevator \\
\hline
Dir = down \\
Requests \cup DownCalls \neq \varnothing \\
CurrentFloor \leq \\
\qquad min\,(Requests \cup DownCalls) \\
CurrentFloor' = max\,\{x : \mathbb{N}_1 \mid x \in \\
\qquad (Requests \cup DownCalls) \\
\qquad \wedge\, x < CurrentFloor\} \\
Requests' = Requests \setminus \{CurrentFloor'\} \\
UpCalls' = UpCalls \setminus \{CurrentFloor'\} \\
DownCalls' = DownCalls \\
(Dir' = down) \wedge (Door' = Door)
\end{array}
$$

However, when all up-calls and requests above the current floor have been serviced and there are still pending calls and requests, the elevator has to start moving downwards (*ChangeUpToDown*). When all down-calls and requests below the current floor have been serviced and there are still pending calls and requests, the elevator has to start moving up (*ChangeDownToUp*).

*ChangeUpToDown*

$$
\begin{array}{l}
\Delta Elevator \\
\hline
Dir = up \\
(Requests \cup UpCalls \neq \varnothing \wedge \\
CurrentFloor > \\
\qquad max\,(Requests \cup UpCalls)) \vee \\
Requests \cup UpCalls = \varnothing \\
Requests \cup DownCalls \neq \varnothing \\
CurrentFloor' = \\
\qquad max\,(Requests \cup DownCalls) \\
Requests' = Requests \setminus \\
\qquad \{CurrentFloor'\} \\
UpCalls' = UpCalls \\
DownCalls' = DownCalls \setminus \\
\qquad \{CurrentFloor'\} \\
(Dir' = down) \wedge (Door' = Door)
\end{array}
$$

*ChangeDownToUp*

$$
\begin{array}{l}
\Delta Elevator \\
\hline
Dir = down \\
(Requests \cup DownCalls \neq \varnothing \wedge \\
CurrentFloor < \\
\qquad min\,(Requests \cup DownCalls)) \vee \\
Requests \cup DownCalls = \varnothing \\
Requests \cup UpCalls \neq \varnothing \\
CurrentFloor' = \\
\qquad min\,(Requests \cup UpCalls) \\
Requests' = Requests \setminus \{CurrentFloor'\} \\
UpCalls' = UpCalls \setminus \{CurrentFloor'\} \\
DownCalls' = DownCalls \\
(Dir' = up) \wedge (Door' = Door)
\end{array}
$$



DOI: 10.1002/smr

When all requests and calls have been serviced, the elevator checks whether there are new calls. If there are up-calls, then it restarts moving upward (*RestartMovingUp*). If there are down-calls, then it restarts moving down (*RestartMovingDown*).

*RestartMovingUp*

$$\Delta Elevator$$

$$Dir = up$$
$$UpCalls \neq \varnothing$$
$$CurrentFloor > max\ UpCalls$$
$$DownCalls \cup Requests = \varnothing$$
$$CurrentFloor' = min\ UpCalls$$
$$Requests' = Requests$$
$$UpCalls' = UpCalls \setminus \{CurrentFloor'\}$$
$$DownCalls' = DownCalls$$
$$(Dir' = up) \land (Door' = Door)$$

*RestartMovingDown*

$$\Delta Elevator$$

$$Dir = down$$
$$DownCalls \neq \varnothing$$
$$CurrentFloor < min\ DownCalls$$
$$UpCalls \cup Requests = \varnothing$$
$$CurrentFloor' = max\ DownCalls$$
$$Requests' = Requests$$
$$UpCalls' = UpCalls$$
$$DownCalls' = DownCalls \setminus \{CurrentFloor'\}$$
$$(Dir' = down) \land (Door' = Door)$$

$$Move == BasicMoveUp \lor BasicMoveDown \lor ChangeUpToDown \lor ChangeDownToUp \lor RestartMovingUp \lor RestartMovingDown$$

The state of the door is also important. It can be opened, and in case of a request or call, it is closed.

*OpenDoor*

$$\Delta Elevator$$

$$CurrentFloor' = CurrentFloor$$
$$Requests' = Requests$$
$$UpCalls' = UpCalls$$
$$DownCalls' = DownCalls$$
$$Dir' = Dir$$
$$Door' = open$$



DOI: 10.1002/smr

*CloseDoor*

$\Delta Elevator$

---

$Requests \cup DownCalls \cup UpCalls \neq \varnothing$
$CurrentFloor' = CurrentFloor$
$Requests' = Requests$
$UpCalls' = UpCalls$
$DownCalls' = DownCalls$
$Dir' = Dir$
$Door' = closed$

When there are no requests the elevator stays at the current floor.

*NoRequestsOrCalls*

$\Xi Elevator$

---

$Requests \cup UpCalls \cup DownCalls = \varnothing$

The elevator repeatedly closes the door, moves, and opens the door. Between two events it may receive passenger events.

$$
\begin{aligned}
&MoveCycle == \\
&\quad (CloseDoor \fatsemi PassengerEvent^{\fatsemi}_{9} \\
&\quad Move \fatsemi (PassengerEvent)^{\fatsemi}_{9} \\
&\quad OpenDoor \fatsemi PassengerEvent)^{*} \\
&ElevatorCycle == \\
&\quad (MoveCycle \lor NoRequestsOrCalls)^{*} \\
&FunctioningElevator == \\
&\quad (PassengerEvent \fatsemi ElevatorCycle)^{*}
\end{aligned}
$$

## APPENDIX C: SRN OF THE ELEVATION SPECIFICATION

Figure C1 shows the SRN of the elevator specification of Appendix B. The graph represents the 32 higher-level primes and their syntactical interrelationship.

The output has been generated by *dotty* (see www.graphviz.org for a description of the tool) and by doing an auto-layout. Thus, no re-arrangement of the vertices has been made.



DOI: 10.1002/smr

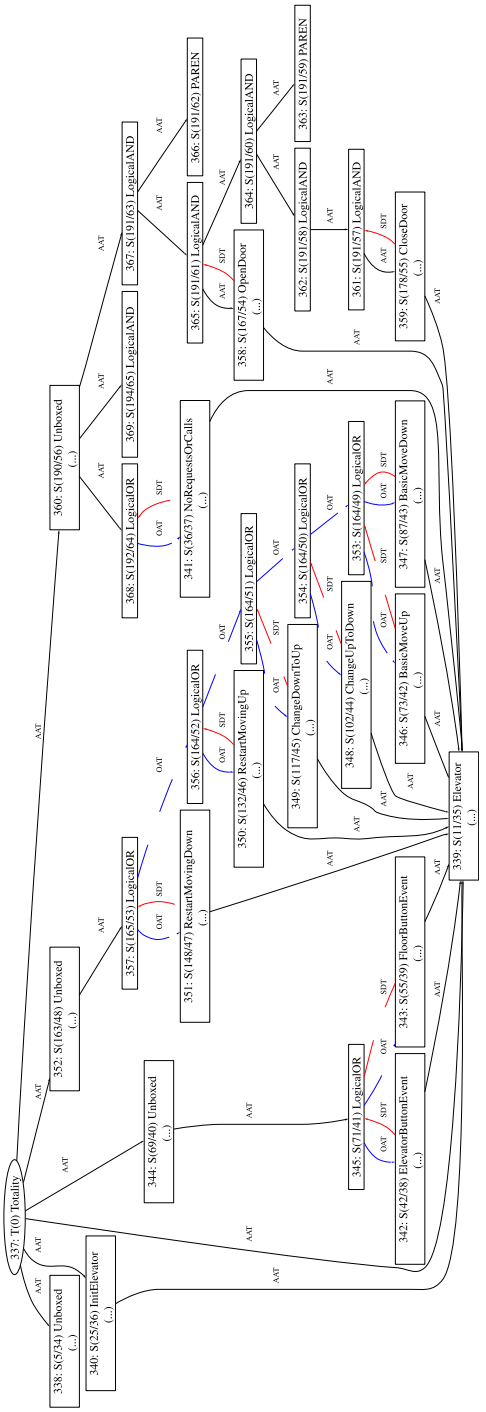


Figure C1. The *Elevator* specification out of [8]. Please note that for reasons of space several dependencies are omitted (logical combinations are resolved by AAT, AND-arc types and OAT, OR-arc types).




DOI: 10.1002/smr

**ACKNOWLEDGEMENTS**

Many thanks to Abdel-Hamid Bouchachia and Marcus Hassler for intensive discussions on clustering and the topic of cluster validation techniques. I am also very grateful for the remarks of Roland Mittermeir and to the anonymous reviewers of the ICSM'06 program committee, whose constructive remarks motivated me to further elaborate that topic.

**REFERENCES**

1. Bowen JP, Hinchey MG. Seven more myths of formal methods. *IEEE Software* 1995; **12**(4):34–41.
2. Hall A. Seven myths of formal methods. *IEEE Software* 1990; **7**(5):11–19.
3. Ross PE. The exterminators. *IEEE Spectrum* 2005; **42**(9):36–41.
4. van Lamsweerde A. Formal specification: A roadmap. *Proceedings of the Conference on the Future of Software Engineering* (*ICSE 2000*). ACM: New York NY, U.S.A., 2000; 147–159.
5. Bollin A. Maintaining formal specifications. *Proceedings 21st IEEE International Conference on Software Maintenance* (*ICSM 2005*). IEEE Computer Society: Los Alamitos CA, U.S.A., 2005; 443–452.
6. Mittermeir RT, Bollin A. *Demand-driven Specification Partitioning* (*Lecture Notes in Computer Science*, vol. 2789). Springer: Berlin, 2003; 241–253.
7. Oda T, Araki K. Specification slicing in a formal methods software development. *Proceedings of the 17th Annual International Computer Software and Applications Conference 1993*. IEEE Computer Society: Los Alamitos CA, U.S.A., 1993; 313–319.
8. Chang J, Richardson DJ. Static and dynamic specification slicing. *Technical Report*, Department of Information and Computer Science, University of California, Irvine CA, U.S.A., 1994; 44.
9. Bollin A. Specification comprehension—reducing the complexity of specifications. *PhD Thesis*, Universität Klagenfurt, Klagenfurt, Carinthia, Austria, April 2004; 311pp.
10. Wu F, Yi T. Slicing Z specifications. *ACM SIGPLAN Notices* 2004; **39**(8):39–48.
11. Spivey JM. *The Z Notation—A Reference Manual* (*C.A.R. Hoare Series*). Prentice-Hall: Upper Saddle River NJ, U.S.A., 1992; 1–23.
12. Bollin A. The efficiency of specification fragments. *Proceedings 11th Working Conference on Reverse Engineering* (*WCRE 2004*). IEEE Computer Society: Los Alamitos CA, U.S.A., 2004; 266–275.
13. Rajlich V, Wilde N. The role of concepts in program comprehension. *10th International Workshop on Program Comprehension* (*IWPC 2002*). IEEE Computer Society: Los Alamitos CA, U.S.A., 2002; 271–278.
14. Kozaczynski W, Ning J, Engberts A. Program concept recognition and transformation. *IEEE Transactions on Software Engineering* 1992; **18**(12):1065–1075.
15. Harandi MT, Ning JQ. Knowledge-based program analysis. *IEEE Software* 1990; **7**(1):74–81.
16. Welty CA. Augmenting abstract syntax trees for program understanding. *Proceedings of the 1997 International Conference on Automated Software Engineering*. IEEE Computer Society: Los Alamitos CA, U.S.A., 1997; 126–133.
17. Quilici A, Yang Q, Woods S. Applying plan recognition algorithms to program understanding. *Automated Software Engineering*: *An International Journal* 1998; **5**(3):347–372.
18. Burnstein I, Saner F, Limpiyakorn Y. Using an artificial intelligence approach to build an automated program understanding/fault localization tool. *Proceedings 11th IEEE International Conference on Tools with Artificial Intelligence* (*ICTAI 1999*). IEEE Computer Society: Los Alamitos CA, U.S.A., 1999; 69–76.
19. Woods S, Quilici A. Some experiments toward understanding how program plan recognition algorithms scale. *3rd Working Conference on Reverse Engineering* (*WCRE'96*). IEEE Computer Society: Los Alamitos CA, U.S.A., 1996; 21–30.
20. Weida RA. Closed terminologies and temporal reasoning in description logic for concept and plan recognition. *PhD Thesis*, Graduate School of Arts and Sciences, Columbia University, New York NY, U.S.A., 1996; 322pp.
21. Wilde N, Scully MC. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance*: *Research and Practice* 1995; **7**(1):49–62.
22. Burnstein I, Roberson K, Saner F, Mirza A, Tubaishat A. A role for chunking and fuzzy reasoning in a program comprehension and debugging tool. *9th International Conference on Tools with Artificial Intelligence* (*ICTAI 1997*). IEEE Computer Society: Los Alamitos CA, U.S.A., 1997; 102–109.
23. Halkidi M, Batistakis Y, Vazirgiannis M. On clustering validation techniques. *Journal of Intelligent Information Systems* 2001; **17**(2–3):107–145.
24. Chen K, Rajlich V. Case study of feature location using dependence graph. *8th International Workshop on Program Comprehension* (*IWPC 2000*). IEEE Computer Society: Los Alamitos CA, U.S.A., 2000; 241–247.



DOI: 10.1002/smr

25. Diller A. *Z—An Introduction to Formal Methods*. Wiley: Baffins Lane, Chichester, U.K., 1999; 3–121.
26. Bezdek JC, Ehrlich R, Full W. The fuzzy C-means clustering algorithm. *Computers and Geoscience* 1984; **10**(2): 191–203.
27. Bowen JP. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press (ITCP): London, U.K., 1996; 302pp.

**AUTHOR'S BIOGRAPHY**

**Andreas Bollin** is an Assistant Professor at the Software Engineering and Soft Computing Group at the Alpen-Adria Universität Klagenfurt. He received his PhD in computer-science from Alpen-Adria Universität Klagenfurt in 2004 and his MSc in software-technology from University of Technology Graz in 1999. His research interests include software comprehension, reverse engineering, concept location, formal specifications, and didactical aspects of e-learning and new technologies. He is member of the IEEE and the ACM.



DOI: 10.1002/smr