Concept Management: Identification and Storage of Concepts in the Focus of Formal Z Specifications

Daniela Pohl and Andreas Bollin

Alpen-Adria Universität Klagenfurt, 9020 Klagenfurt, Austria
{daniela,andi}@isys.uni-klu.ac.at
http://www.uni-klu.ac.at/tewi/inf/isys/sesc/index.html

Abstract. Concept location is a necessary but all too often laborious task during maintenance phases. Part of the reasons is that repeatedly the same or similar concepts have to be reconstructed, which is a resource and time-consuming process. This contribution investigates the situation and suggests a framework that persistently stores conceptual elements and their dependencies in an *SQL* database. On the example of formal Z specifications it demonstrates that concept location is alleviated by simple queries that automatically identify concepts based on the database entries.

Keywords: Concept location, Formal Z specifications, Maintenance.

1 Introduction

"*People like to write code, but they do not like to read somebody else's code.*" This statement becomes increasingly apparent as the number of software systems in use is growing – and have to be maintained. Why might this be the case?

In [1, p.242] it is postulated 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 expressions formulated in low level code. This observation is above all true when the text or code expresses a concept previously unknown to the reader – which is often the case in maintenance situations. In the lucky case there are at least high-level specification documents around, but, without supporting tools, the identification of the relevant information stays a hard business.

Maintenance activities are often formulated in terms of adding/changing/deleting features or concepts [2], and concept location techniques play an important role, in software as well as in specification maintenance. Formal specification frameworks provide excellent support for editing and verification, but they do not provide concept location facilities. A (semi-) automatic identification of concepts is missing and, for formal specifications, also the possibility to store the, often cumbersome, reconstructed concepts to be found in the documents.

The objective of this contribution is to demonstrate that not so much has to be done in order to identify *and* store concepts. We introduce a generic model that is able to deal with documents and concepts of different types. As a proof of concept a prototype for formal Z specifications [3] has been implemented. But the basic ideas also apply to other artifacts ... from natural language descriptions to program code.

The paper is structured as follows. Sec. 2 discusses the related work. Sec. 3 derives the requirements for a framework that is able to persistently store identified concepts. With the necessary basics in concept location of Sec. 4 in mind, Sec. 5 introduces the key ideas behind our suggested framework. Sec. 6 evaluates its functionality in respect to correctness and time complexity. The contribution closes with a short summary and an outlook of work to be done.

2 Related Work

Due to the size of today's systems, maintenance and reverse engineering activities are usually supported by tools and frameworks.

At first, there are SW-Engineering frameworks that can also be used for reverse engineering activities [4,5,6,7,8]. Usually, they enable the reconstruction or extractions of diagrams from code, and thus establish links between different representations supporting round-trip engineering. Their disadvantage is that they are limited to a very specific notation (e.g. UML) or assume that the code has already been written within the framework.

Another group is that of explicit reverse engineering tools. There, the input is the code or an abstract representation of it, and they sustain the process of creating extractions or views onto the source. Popular representatives are RIGI [9], going back to the work of Müller, Tilley, and Wong in the early 90s, or Bauhaus [10]. In the meantime there are a lot of extensions and, especially for C++ and Java programs, similar frameworks [11,12,13,14,15,16].

Finally, there are frameworks that focus explicitly on concept location [17,18,19]. They make use of techniques similar to those of reverse engineering environments, but provide additional support for storing and retrieving previously identified concepts.

Environments for formal specifications have their focus on writing down a syntactically correct specification and providing verification support. They are not meant to be used for reverse engineering. However, one tool-set that permits looking at a specification from different angles is *VDMTools* with its *RoseLink* feature [20]. It can be used to generate UML diagrams from VDM specifications. Tools for *B* also focus on the creation of the specification. Some representatives, e.g. Atelier-B [21], at least provide views onto dependencies between the components. In the case of Z the situation is almost the same. One exception is the *ViZ* toolkit [22], where the emphasis was laid on concept location. But *ViZ* also has its limitations, first and foremost the inability to persistently store identified concepts.

3 Maintenance Support

The motivation for this work goes back to a project where we tried to improve maintenance and re-engineering activities of formal Z specifications. A big advantage of formal specifications is their semantic density. One can express his or her thoughts precisely and on a high level of abstraction. But with that, the complexity (and density) of even small specifications is quite high. As shown in [23], specifications might have thousands of dependencies between their elements, and comprehension aids are definitely necessary.

3.1 RE of Formal Z Specifications

At first sight approaches from the traditional field of software comprehension are not suitable. Z (among others) is a state-based, declarative specification language, with no explicit control and data flow – dependencies that are typically utilized when looking for concepts. But there is a solution to this problem.

In [24, p.60–63] a syntactical approximation to the identification of dependencies was described, which then enabled the identification of concepts like slices, chunks, and clusters within formal specifications. *ViZ* (for *Vi*zualization of *Z* Specifications) implements these algorithms and supports typical comprehension activities. But with its employment the following issues have been observed:

– The same comprehension steps are often carried out more than once - even if there is additional documentation. So dependencies and concepts identified once have to be reconstructed again.
– The calculation and identification of concepts is still a time consuming task. Moreover, these calculations and findings are lost when the framework is closed and the state is not saved for later use.
– It is not sufficient to look at a concept in isolation. Depending on the problem at hand more than just a single view onto the artifact has to be generated.

To summarize, a framework sustaining comprehension tasks should not ignore the above observations. It should support the identification of new concepts at different levels, enable the linkage between them, and be able to store the findings persistently in a database. Please note that the observations above are not only limited to the field of Z specifications. They apply to other artifacts, too. Due to the resource-consuming calculations necessary for dense and complex formal specifications, the storage of concepts is of major importance in our case.

3.2 Multi-dimensional Problem

The reconstruction of concepts within an artifact is not trivial. In order to reconstruct (or better approximate) the concepts a former developer had in mind, one has to take into account that different facets led to the writing of the artifact:

– *The environment and context of the problem.* There are maybe several assumptions the developer had to consider and that are not fully documented. So, an artifact only makes sense when put into the right context.
– *The concepts inherent in the language.* Different (programming) languages are differently suitable for describing problems. In fact, the semantics behind a language is often used to reduce writing effort. E.g. dependencies do not have to be made explicit, names are declared once, and it is clear when they are usable and when not. Those concepts, let us call them *"behind the scenes"*, are important for grasping the whole meaning.

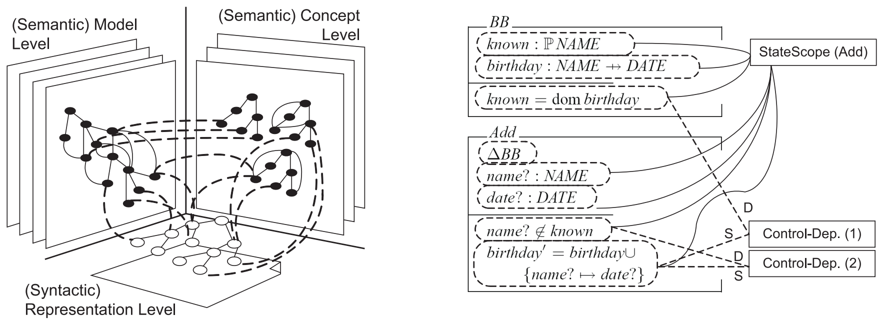

Fig. 1. (Left) An artifact contains concepts in three different dimensions and at different levels. (Right) Specifications are dismantled into syntactical elements (primes) and then extended by dependency and scope annotations.

– *Concepts made explicit in the source.* The concepts mentioned above are problem- and language-inherent. What is left are those concepts that are visible in the artifact. E.g. a case-statement represents an n-ary decision, and its meaning is clearly defined. Such concepts are called *"before the scenes"*.

When one is at least familiar with the problem field and the environment, the identification of the concepts behind and before the scenes might be seen as a multidimensional problem. Fig. 1 (left side) demonstrates this viewpoint.

The **(Syntactic) Representation Level.** At first, there is the artifact itself. It has been written in some pre-defined language, with clearly defined rules for its syntax. It is assumed that it can be divided into a structured set of basic elements (primes, statements, paragraphs ...). This sequence of elements, its nouns and verbs, and the structure make up a lot of the underlying concept(s). Thus, the representation level deals with the source and the structure of the artifact.

The **(Semantic) Model Level.** In general, a language comes along with a clearly defined semantics (e.g. statements have to be put into some order). This implies a specific meaning and leads to several dependencies and relations between the elements (see Fig. 1). These rules are not written down in the artifact, but belong to it and make up another part of the underlying concept(s). They can be seen as named concepts, going back to the semantic possibilities of the language at hand.

The **(Semantic) Concept Level.** What is left are the concepts the developer expresses unconsciously. They describe specific aspects of the problem and are recognizable when looking at the artifact from some distance. These mental macros, as Baxter et. al. [25] call them, express higher-level concepts, and program comprehension techniques are typically used to carve them out. To this dimension belong concepts like slices [26], chunks [27], clichés [28], and different types of clusters [29].

To exemplify the situation, the calculation of a specification or program slice (stored in the concept level) depends on the concept of dependencies (model level), the concept of scope (model level), and the basic elements in the source (representation level). When these concepts (at different levels) are calculated they can be stored in a database for later use.

4 Formal Specification Concepts

The model presented above has been mapped to a database schema and forms the basis for the concept location process. Though the strength of the framework is to deal with different types of documents, our experiences arise from the scope of maintaining formal Z specifications – which also was the starting point of the requirements' considerations. The specification concepts we are interested in are those as described in more detail in [22]: slices, chunks, and clusters.

4.1 Conceptual Elements

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.

As explained in more detail in [22, p.81], the basic elements such concepts are built upon are called specification primes. Such *formal specification primes* (also called prime-objects) also represent the basic entities of a specification. They are built from literals of the specification and form logic, syntactic, or semantic units. A prime is a syntactically coherent sequence of literals within a specification, forming semantic entities that can be paraphrased by a short sentence in natural language.

With programming languages, primes would be programming statements. In the case of formal Z specifications these primes are declarations, definitions, and predicates. Fig. 1 (to the right) marks the primes by dashed ellipses, e.g. the prime "$name? \notin known$" (the second prime from below).

When primes are combined they do form so-called *higher-level primes*. The *Add* schema operation in Fig. 1 is an example of such a higher-level prime, telling the user about the things happening when the operation applies.

4.2 Specification Concepts

Concepts within formal specifications are identified in an iterative manner [22, p.83]. Starting with a domain-level request, one forms a mental model of the problem in mind and concept location is about to begin. Concept candidates are identified and matched against the model of the problem. When the candidates match, the concept is (very likely) identified and the elements of the related candidates are tagged. The concept location process makes use of the following steps: pattern matching, slicing and chunking, and cluster identification.

As explained in [30], experienced users first browse the text and try to identify relevant parts by *grep*-ing for keywords. When this is not successful, more complicated methods are used. Structures are especially of interest, and clustering is a feasible way in identifying related regions. The selection might be based on the use of identifiers, or on the number of dependencies that glue the primes together. Similarly, slices and chunks can be generated for a point of interest by just looking at specific primes and by following different types of dependencies.

Specification concepts are identified by looking at dependencies among primes. For the calculation of slices, chunks, and clusters, control- and data-dependencies are needed, and though these dependencies are not explicitly available, they can be reconstructed by looking at pre- and post- conditions. The approach goes back to the work of Oda and Araki [31] and has been refined in [32,24]. The basic idea is that, within a specific scope, primes that are part of post-conditions are dependent on primes that contribute to pre-conditions. In order to ease their identification, all primes get tagged with annotations. For every identifier used in a prime the following meta-information is assigned to the prime: CI (channel input) when it is an input identifier which is decorated by a $\boxed{?}$, CO (channel output) for an result identifier decorated by an $\boxed{!}$, D (declaration) for an identifier that denotes the identifier's after-state and which is decorated by a $\boxed{'}$, T (type declaration) for identifiers that are declared, and U (used) otherwise. So, the two Z primes (of the *Add* schema in Fig. 1)

$$P1: \quad name? \notin known$$
$$P2: \quad birthday' = birthday \cup \{name? \mapsto date?\}$$

would be tagged as follows. Prime $P1$ is annotated by $\{CI \mapsto \{name\}, U \mapsto \{known\}\}$, and prime $P2$ is annotated by $\{D \mapsto \{birthday\}, U \mapsto \{birthday\}, CI \mapsto \{name, date\}\}$. Post-condition primes are those primes that have a tag containing a D or CO set. In our case $P2$ would be a post-condition prime, prime $P1$ is a so-called pre-condition prime.

The identification of dependencies is explained in more details in [24, pp.126–132]. However, when the meta-information is stored in the database (and assigned to the prime objects), the queries are quite simple. Our agents, as introduced in Sec. 5.2, make use of this meta-information in form of *SQL* queries.

5 Concept Location Framework

The framework for identifying the different concepts in Z specifications is designed to cope with different types of artifacts. It implements a traditional client-server architecture pattern based on the *EJB* Technology. The client is responsible for visualizing the results and for triggering the concept extraction. On the server side it is designed to handle different types of artifacts. Whenever a document is stored, different analysis tasks are started by a scheduler extracting concepts, and the findings are stored in the database again (see Sec. 5.2).

5.1 Database

The database forms the basis for the management of conceptual elements and their dependencies, sustaining the concept location process of formal Z specification documents.

There are four areas covered by the database. Three of these areas are related to the multi-dimensional view as described in Sec. 3.2. The forth area is used for the management of artifacts within the software engineering life-cycle.

Management/Project Pane. Based on the software engineering process, the *Management/Project* section deals with the management of artifacts within different phases of

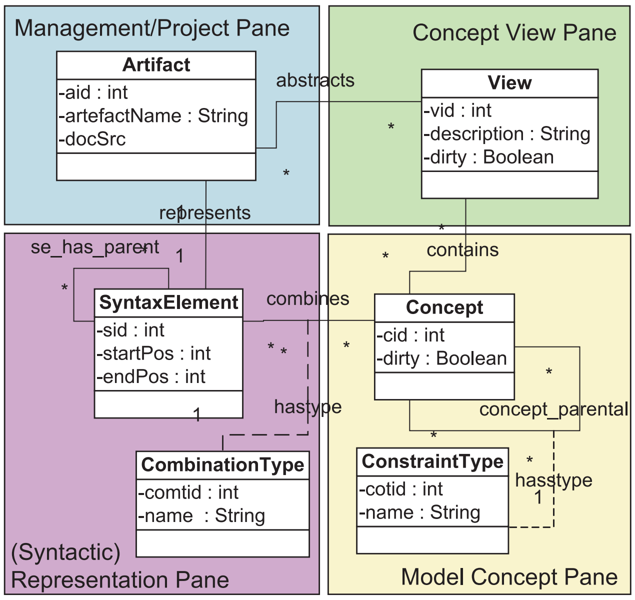

Fig. 2. The four different panes of the database model (Please note that for reasons of space only the major entities are shown. See [33] for more details.)

the project. There, a *Project* consists of different *Phases*. Within each phase *Artifacts* are created, most of them depending on each other. Different artifact versions might exist. Hence, the database schema takes this into account by assigning the *ArtifactMetaData* information to an artifact.

(Syntactic) Representation Pane. Every artifact, independently of its nature, consists of a certain structure. This structure is built upon so called *SyntaxElement*s. *SyntaxEle– ments* are characterized by their *ElementType*s:

- *Content:* It represents a pure structural element (so-called basic elements like sentences, expressions, or statements).
- *Presentation:* Text is often decorated (e.g. by boxes). As sometimes this decoration carries information, it is also stored.
- *Aggregation:* It provides the possibility to explicitly mark higher-level concepts that have been created by the aggregation of basic elements.

Syntax elements carry a lot of information. E.g. they refer to identifiers, define labels, or describe some input operations. A set of meta-data is introduced to store them. Every data entry of *ElementMetaData* belongs to a specific *AnnotationType*, and so different (but consistent) categorizations get possible.

Model Concept Pane. A *Concept* corresponds to one or several syntactical elements (*SyntaxElements*). For different types of concepts also different relationships are possible. This is done by the *CombinationType* entity. Besides, it is possible to express some kind of direction or ordering between the related elements. The characterization of a concept is made up by the *ConceptType* entity. Concepts also form hierarchies, and to express these relations, an n-to-m recursive relationship is introduced.

Concept View Pane. The database schema allows for different views onto the concepts, be them explicitly or implicitly available. The main purpose of the *View* entity is to cluster related concepts (concepts of the same type) or to form different views onto the current artifact. This information is, besides the creation date, stored in the *ViewMetaData* entity. Every view belongs to a certain category. This classification is stored in the *ViewType*.

It is also possible to annotate a view with *ViewData* entries of specific *ViewDataTypes*. Those entries are, e.g., used to store metrics of clusters or other characteristics relevant for concepts within the view. These steps are carried out by agents like those introduced in the following section.

5.2 Agents

Our prototype provides different agents: scope agents that regard scope rules of Z, dependency agents for reconstructing dependencies, and, based on them, slice/chunk/cluster agents for carving out higher-level concepts.

For the creation of slices or chunks typically two different types of dependencies (data-, and control-dependencies) and the syntactical environment are necessary. So, at first, these dependencies have to be extracted, but the extraction is complicated due to language-specific scope rules. The first task for our framework is therefore to reconstruct the concepts representing the scope.

In the context of formal Z specifications three types of scopes can be identified (and are calculated by three agents in our framework). The *Declaration Scope* represents all visible declarations for a prime in the specification. The scope is also needed to derive the syntactical dependencies and thus for building syntactically correct partial specification. The *State Scope* deals with schema inclusions within a specification document and aggregates the primes of the inclusion and the primes of the including schemata (see Fig. 1, $Add_{(State)}$ for an example). Finally, the *Connectivity Scope* merges all primes of two or more schemata combined via schema operations.

In our framework, at first, the agents launch queries to identify the correct scopes, then they start reconstructing control- and data dependencies. Sec. 5.3 demonstrates the simplicity on the example of control dependency identification.

After scope and dependency calculation the *Slice* and *Chunk* agents can be activated by the user. Beginning with a ”point of interest” (a set of primes), the agents calculate slices and chunks by following the stored control- and data dependencies. The results are again stored in the database for later use.

The last agent presented here is called *Clustering Agent*. It is responsible for the generation of clusters of a specification document. In order to ease deciding about the most useful number of clusters to be generated, the agent pre-calculates and stores all variants of them. Every cluster view is then extended by meta-information. This meta-data describes different types of cluster-metrics, like the partition entropy or the partition coefficient measure. This information can later help the user to decide about the usefulness of the clusters.

Some of the agents are executed in parallel; other agents have to wait. Therefore all agents are registered in an agent scheduler, which is responsible for the right execution order.

5.3 Queries for Concept Location

Our framework makes use of a simple idea: the calculation logic is moved from traditional program code to SQL queries disposed by the agents. The extraction is done by expressions which are based on the annotations of the primes in the database. For Z documents the necessary queries are already implemented.

To demonstrate the elegance of the queries we look at the steps necessary to carve out control dependencies from Z specifications. The relevant primes in the database are the syntax elements tagged by the *Content* element type. The extraction-process then uses the *State* and *Connectivity Scope* for the calculation.

$$\begin{aligned}&\Pi_{sid}\ \sigma_{AnnotationType.name="D"}\\&((\sigma_{Concept.id=act}Concept \bowtie\\&(\sigma_{ConceptType.name="State"}\,ConceptType))\\&\bowtie SyntaxElement \bowtie\\&ElementMetaData \bowtie AnnotationType)\end{aligned} \tag{1}$$

$$\begin{aligned}&\Pi_{sid}(\sigma_{Concept.id=act}Concept \bowtie\\&(\sigma_{ConceptType.name="State"}\,ConceptType))\\&[sid \neq sid]\\&\Pi_{sid}\\&(\sigma_{\substack{AnnotationType.name\neq"T"\ or\\ AnnotationType.name\neq"C"\ or AnnotationType.name\neq"D"}}\\&SyntaxElement \bowtie ElementMetaData\\&\bowtie AnnotationType)\end{aligned} \tag{2}$$

The queries (1) and (2) above extract control–dependencies of the *Add* schema operation of the *'Birthday Book'*-Specification (where *act* represents the identifier of the current *State Scope*). The queries lead to the source (*S*) and destination (*D*) primes for the dependency arcs. In fact, the results of the queries are elements of the *SyntaxElement* entity. The agent takes all elements resulting from the first query and connects them to the resulting elements of the second query. Every pair forms a *Concept* within the database. This information is stored in the database and results in the concept entries *Control-Dep. (1)* and *Control-Dep. (2)*) as exemplified in Fig. 1.

The identification of data–dependencies is similar to that of control–dependencies. Its only difference is related to the *U* tag, and the consideration of the name of an identifier. A detailed description of the queries for scope and dependency calculation can be found in [33, p.102-106].

Table 1. Complexity attributes and calculation time (in seconds)

Specification	Lines	Pages A4	Primes	CD	DD	ViZ [s]	EJB/DB [s]	Concepts
BB	40	2	34	10	5	4.6	7.0	36
Cinema	95	4	74	121	43	75.3	43.2	114
Petrol	88	3	65	192	177	152.9	51.9	219
Elevator	193	6	185	1,628	992	1,223.4	709.3	984

6 Evaluation

The evaluation of the framework was carried out in two steps. First, the correctness of the identified concepts were checked, and, secondly, the usefulness in respect to performance explored. In fact, both steps also hearken back to results we gained from the existing *ViZ* framework.

6.1 Setting and Correctness

The first step was the validation of the concepts that have been identified by the agents and stored in the database. The evaluation involves specifications of raising sizes, known as Birthday Book [3], Petrol Station [24], and Elevator [32]. Additionally, a student's specification (Cinema) was added to the set. Tab. 1 (left part) summarizes the key attributes in order to assess the complexity of the specifications. It exemplifies the number of lines, pages (when printed), primes, control- (CD), and data dependencies (DD).

The correctness of the identified concepts was checked in two steps. At first, the new framework was used to identify dependencies, slices, and chunks. The results were then exported to a structured file. In a second step these entries have been compared to the concepts and dependencies identified by the *ViZ* framework. As Tab. 1 (right column) demonstrates, this involved 1353 concepts (consisting of slices and chunks for every prime occurring in the predicate part of every schema) and 3168 dependencies (CD and DD).

6.2 Performance Considerations

As every dependency and concept has been detected correctly, we were also eager to see whether the framework scales and improves operating speed. In fact, in our case

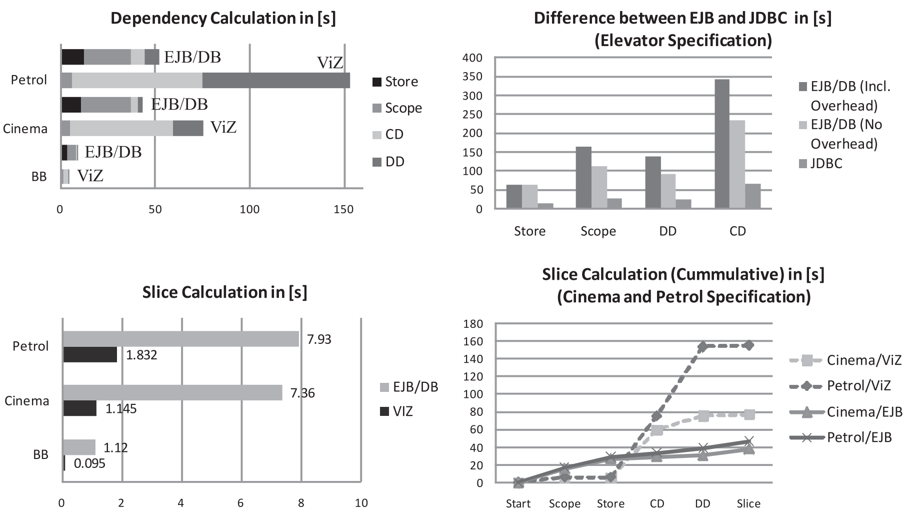

Fig. 3. Performance considerations between the ViZ and the EJB framework

operation time it up to (a) dependency calculation, (b) storage and retrieval, and (c) concept identification.

In the case of *ViZ* the calculation of dependencies (and thereinafter slices or chunks) is time-consuming. *ViZ* uses an annotated graph to store primes and its connections, and dependency calculation is based on reachability considerations. It has a runtime performance of $O(n * (m + n * log * n))$ (with n related to the number of primes and m related to the number of dependencies in the specification). The new framework considers Def-Use equations based on scope relations (that are stored in the database), and its runtime complexity is in $O(n^2)$. Tab. 1 (center part) presents the time needed to calculate all dependencies for the *ViZ* environment and the *EJB* based framework (where the system consisted of a Windows XP Professional OS, Intel Core2 CPU, 2.00GHz, 2 GB RAM). This difference can also be seen in Fig. 3 (top left). Though *ViZ* does not store the elements in a database, the total time is much higher due to the extra time needed for control and data dependency calculation.

The type of the database access is also crucial for the performance. The most complex artifact in our considerations is the *Elevator* specification, and it takes about ten minutes till all dependencies are analyzed (and about 2,600 data-sets are stored for later use). As a few thousand data-sets are not so much for a database, we were eager to know why it took so long.

It turned out that a lot of time is lost due to *EJB*'s synchronization between the database and Java's objects. The overhead is about *one-third* of the time. Furthermore, there is very high execution time latency between *EJB* and its corresponding *JDBC* queries. As explained in more details in [34, p.234], in our setting *JDBC* scales about *six times* better than *EJB*. Fig. 3 (top right) demonstrates this time-differences on the example of the elevator specification.

The calculation of concepts like slices or chunks implies looking at a specific prime and following the relevant dependencies. Fig. 3 (bottom left) shows that *ViZ* is definitively faster than the new environment. There, all possible slices for three different specifications have been generated once and the total time measured. *ViZ* is much faster, which is not surprising as it's internal graph already contains the dependencies as arcs and they do not have to be read from a database. However, the new framework stores the slice as a view for later use, and calculated once, it does not have to be (re-)calculated again.

Considering the above observations, the new framework seems to be an improvement in the case of concept location environments for Z specifications. Fig. 3 (bottom right) demonstrates this by accumulating the time till all possible slices have been calculated once. *ViZ* is faster at the beginning, as it does not store the elements in a database, but the new framework invests in storing the syntactical elements in the database and assigns scope information to it. This investment pays back when dependencies are to be calculated, and it outpaces *ViZ*. Retrieving the concepts then is slower, but merely depends on the number of elements to be retrieved by a select operator. In addition to that, they have only to be retrieved once, as after retrieval they are stored as a view in the database. Here the strengths of a relational database pay off.

Though the new framework is faster, we conclude from the analysis above that the use of *EJB* is less suitable. It brings maintenance advantages, but, as also addressed in

[34], one has to expect performance loss that should not be neglected. For this reason we are currently working on a new release of the framework that replaces the middleware technology by *JDBC*.

7 Conclusions

This paper introduces the problem of concept location within state-based specifications and motivates for a framework that persistently stores concepts for later use and fast access. Starting with a thorough analysis of concept location aspects, a database schema has been developed which, thereinafter, forms the basis for our concept location framework for formal Z specifications.

The paper then introduces the key ideas behind our prototype. Besides storing concepts, it is based upon the idea of individual agents that quickly identify different relations among syntactical elements (of our specification) stored in the *MySQL* database. Their elegance originates from the fact that an *SQL* database is very efficient in looking for specific relations between elements, and thus most of the calculation logic could be put into slim *SQL* queries.

The evaluation is based on a comparison with *ViZ*, an already existing concept location framework for Z specifications. The evaluation shows that it produces the same results than *ViZ*, but calculation times varied. The performance of the framework was strongly influenced by *EJB*. The analysis of *JDBC* and *EJB* shows a high factor of performance loss when using *EJB*. *JDBC* scales about six times better than *EJB* in terms of runtime. Additionally, *EJB* implements an intermediate layer and, therefore, runs into performance latencies. It is going to be replaced by *JDBC* in the next release of our framework.

References

1. Mittermeir, R.T., Bollin, A.: Demand-driven Specification Partitioning. In: Proceedings of the 5th Joint Modular Languages Conference (2003)
2. Wilde, N., Scully, M.C.: Software Reconnaissance: Mapping Program Features to Code. Journal of Software Maintenance: Research and Practice 7, 49–62 (1995)
3. Spivey, J.: The Z Notation: A Reference Manual, 2nd edn. Prentice Hall International, Englewood Cliffs (1992)
4. Rational-XDE: IBM Rational XDE DeveloperWorks Home Page, `www.ibm.com/developerworks/rational/products/xde/` (Page last visited: March 2009)
5. Eclipse-GMT: Homepage, `http://www.eclipse.org/gmt/` (Page last visited: March 2009)
6. Nickel, U., Niere, J., Wadsack, J., Zündorf, A.: Roundtrip Engineering with FUJABA. In: Ebert, J., Kullbach, B., Lehner, F. (eds.) Proceedings of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany (August 2000)
7. Jouault, F.: Loosely Coupled Traceability for ATL. In: Proceedings of the European Conference on Model Driven Architecture (ECMDA 2005), Workshop on Traceability (2005)
8. MetaEdit+: Homepage, `www.metacase.com` (Page last visited: March 2009)

9. Müller, H.A., Tilley, S.R., Wong, K.: Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project. In: CASCON 1993, October 1993, pp. 217–226 (1993)
10. Koschke, R.: Software Visualization for Reverse Engineering. In: Diehl, S. (ed.) Dagstuhl Seminar 2001. LNCS, vol. 2269, pp. 524–527. Springer, Heidelberg (2002)
11. Ferenc, R., Beszedes, A., Tarkiainen, M., Gyimothy, T.: Columbus – Reverse Engineering Tool and Schema for C++. In: IEEE International Conference on Software Maintenance, Montreal, Canada, pp. 172–181 (2002)
12. Korshunova, E., Petkovic, M., van den Brand, M.G.J., Mousavi, M.R.: CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code (Tool Paper). In: Working Conference on Reverse Engineering (WCRE 2006), Benevento, Italy (2006)
13. Computer Human Interaction and Software Engineering Lab (CHISEL): SHriMP Homepage, `www.thechiselgroup.org/shrimp` (Page last visited: October 2008)
14. Holt, R.: PBS – The Portable Bookshelf Homepage, `http://www.swag.uwaterloo.ca/pbs/intro.html` (Page last visited: October 2008)
15. Ebert, J., Kullbach, B., Riediger, V., Winter, A.: GUPRO – Generic Understanding of Programs – An Overview. Electronic Notes in Theoretical Computer Science 72(2) (2002)
16. Holt, R., Schürr, A., Sim, S.E., Winter, A.: GXL Graph Exchange Library Homepage, `http://www.gupro.de/GXL/` (Page last visited: April 2008)
17. Chen, K., Rajlich, V.: RIPPLES: Tool for Change in Legacy Software. In: IEEE International Conference on Software Maintenance, p. 230. IEEE Computer Society, Los Alamitos (2001)
18. Xie, X., Poshyvanyk, D., Marcus, A.: 3D Visualization for Concept Location in Source Code. In: Proceedings of 28th IEEE/ACM International Conference on Software Engineering (ICSE 2006), May 20–28, pp. 839–842 (2006)
19. Poshyvanyk, D., Marcus, A.: Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In: Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC 2007), June 26–29, pp. 37–48 (2007)
20. Agerholm, S., Larsen, P.G.: Applied Formal Methods – FM-Trends 98. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 326–339. Springer, Heidelberg (1999)
21. Engineering, C.S.: The Atelier-B Homepage, `http://www.atelierb.eu/index-en.php` (Page last visited: June 2009)
22. Bollin, A.: Concept Location in Formal Specifications. Journal of Software Maintenance and Evolution: Research and Practice 20(2), 77–104 (2008)
23. Bollin, A.: The Efficiency of Specification Fragments. In: Proceedings of the 11th IEEE Working Conference on Reverse Engineering (2004)
24. Bollin, A.: Specification Comprehension – Reducing the Complexity of Specifications. PhD thesis, Universität Klagenfurt (April 2004)
25. Baxter, I.D., Yahin, A., Moura, L., SantAnna, M., Bier, L.: Clone Detection Using Abstract Syntax Trees. In: Proceedings of the International Conference on Software Maintenance, pp. 368–377. IEEE Computer Society, Los Alamitos (1998)
26. Weiser, M.: Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. PhD thesis, University of Michigan (1979)
27. 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 1997, 9^{th} International Conference on Tools with Artificial Intelligence, November 1997. IEEE press, Los Alamitos (1997)

28. Broad, A., Filer, N.: Applying Case-Based Reasoning to Code Understanding and Generation. In: Proceedings of the Fourth United Kingdom Case-Based Reasoning Workshop (UKCBR4), University of Salford, Salford, England, September 1999, pp. 35–48 (1999)
29. Wiggerts, T.: Using Clustering Algorithms in Legacy System Remodularization. In: Proceedings of the 4th Working Conference on Reverse Engineering (WCRE 1997). IEEE Press, Los Alamitos (1997)
30. Rajlich, V., Wilde, N.: The Role of Concepts in Program Comprehension. In: International Workshop on Program Comprehension, pp. 271–278. IEEE Computer Society, Los Alamitos (2002)
31. Oda, T., Araki, K.: Specification slicing in a formal methods software development. In: Seventeenth Annual International Computer Software and Applications Conference, November 1993, pp. 313–319. IEEE Computer Socienty Press, Los Alamitos (1993)
32. Chang, J., Richardson, D.: Static and Dynamic Specification Slicing. In: Proceedings of the Fourth Irvine Software Symposium, Irvine, CA (April 1994)
33. Pohl, D.: Specification Comprehension – Konzeptverwaltung am Beispiel zustandsbasierter Spezifikationen (in German). Master's thesis, University of Klagenfurt, Software Engineering and Soft Computing (Juli 2008)
34. Pohl, D., Bollin, A.: Database-Driven Concept Management – Lessons Learned from Using EJB Technologies. In: 4th International Conference on Evaluation of Novel Approaches to Software Engineering (May 2009)