DATABASE-DRIVEN CONCEPT MANAGEMENT
Lessons Learned from Using EJB Technologies

Daniela Pohl, Andreas Bollin
Software Engineering and Soft Computing, Klagenfurt University, Universitätsstrasse 65-67, Klagenfurt, Austria
daniela@isys.uni-klu.ac.at, andi@isys.uni-klu.ac.at

Keywords: EJB-Technologies, Concept Location, Formal Methods

Abstract: The identification of concepts in artifacts is a necessary task during software maintenance activities. However, the concept location process is laborious, and, for some types of artifacts, also often impeded. This paper therefore presents an approach that is able to identify and store concepts quickly and on the basis of simple SQL queries. The ideas are then evaluated on formal Z specifications. Finally, this contribution demonstrates that the suggested use of EJB-technologies redound to more time-complexity than necessary.

1 INTRODUCTION

As developers we are always surrounded by complexity. Partly, this is because our applications have to get more sophisticated. Partly, this is because also our designs get more and more complex in order to fulfil these (sometimes even conflicting) objectives. With it the related design documents explode in size and imply complications. This situation was already reflected by C.A.R. Hoare in the early 1980s, who stated that *[..] there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies* (Hoare, 1981). Small documents are rather exceptions, and locating deficiencies is a major business for a software personnel. The bad news is that fixing those parts is impeded by the very mentioned problems of size and complexity.

As scientists we are therefore faced with the challenge to overcome at least parts of these hurdles. As a first step we have to assist the developer in understanding the relevant parts of the system and the related maintenance activities. As a second step we have to ensure that the relevant parts of the software (to be changed) can be located easily. These tasks are nowadays supported by software comprehension environments (Nickel et al., 2000; Jouault, 2005; Borland, 2008; Eclipse, 2008), reverse engineering frameworks (Müller et al., 1993; Burnstein et al., 1997; Ebert et al., 2002; Ferenc et al., 2002; Korshunova et al., 2006), and concept location plugins (Chen and Rajlich, 2001).

Since several years our working group tries to enhance the situation by supporting reverse engineering in all its aspects (Mittermeir et al., 2001). This also includes reverse engineering frameworks for formal methods (Bollin, 2004) or, recently, rule-based systems (Wakounig, 2008). Our approaches make use of the identification of relationships and the reconstruction of concepts within a system – with the goal to exemplify them to the user. However, to profit from these findings later on, these concepts have to be (inter-)connected to the different artifacts and stored for future use. In the course of a master thesis (Pohl, 2008) this approach has now been evaluated.

In order to share our experiences, the paper is structured as follows: Sec. 2 introduces the notion of concepts and their detection in formal Z specifications. Sec. 3 then presents the architecture of the framework and the related database model. Sec. 4 is dedicated to the use of Enterprise Java Beans (EJB). Sec. 5 describes the evaluation steps and lessons learned. Finally, Sec. 6 concludes this contribution with a short summary.

2 CONCEPTS

The identification of concepts is a time-consuming process. Therefor, the objective of this contribution is to evaluate the identification and storage of concepts by the use of a relational database. In light of the considerations in Sec. 1, a generic DB model (for storing and connecting concepts) and a framework (for dealing with them) have been developed. Their major task is to support the (semi-)automatic identification of concepts. However, first it has to be clear what we meant by the notion of a concept, and which artifacts are taken as experimental subjects.

2.1 Concepts

When trying to understand a system, concepts are generally seen as *perceived regularities in events or objects, or records of events or objects, designated by a label* (Rajlich and Wilde, 2002). One is looking for related parts and trying to assign a name/meaning to them. Additionally, by aggregation, new (abstract) concepts can be built. The concepts we are looking for are exactly those parts with dependencies within and across artifacts.

2.2 Z Specifications

In order to demonstrate generality, we decided to focus on artifacts that are at a very high abstraction level and that are inherently complex: formal Z specifications (Spivey, 1989). They seem to be most useful as they are semantically very compact and are of a declarative nature. This implies that dependencies are definitely hard to identify, and the assumption is that other artifacts (like program text) will not complicate the situation.

Concept identification within formal specifications depends on the notion of control and data dependencies between their basic building bricks (also called *prime elements*). Their calculation is impeded by their declarative nature, but (with some limitations) they can be reconstructed. Basically, this is done by regarding scope rules and looking at the primes' identifiers and by assigning definition (D) declaration (T) and use (U) tags to them. Primes that describe an after state (in our case they contain at least one D-tag) are said to be control dependent on primes that do not describe such an after state. When also regarding a specific identifier within the primes, data dependencies can be detected [1].

[1]For an in-depth discussion on dependencies and concepts see (Bollin, 2008).

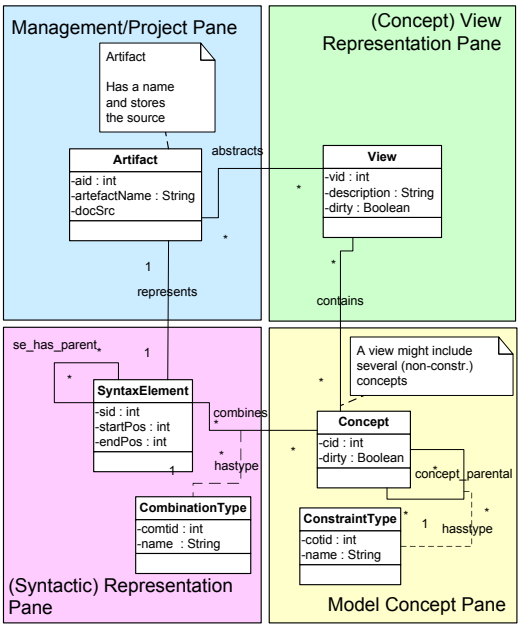

Figure 1: Abstraction of the concept database.

Based on the identified dependencies, the following partial specifications can be defined: specification slices, chunks, and clusters. These specification abstractions are hereinafter treated as concepts that are to be identified *via* and stored *in* the database. The following section introduces the architecture of the framework and the related database schema.

3 ARCHITECTURE

The framework for identifying the different concepts in Z-Specification is based on the architecture shown in Figure 2. This architecture is designed in such a way to extend the functionality for other different document types, like program code, also necessary in the software engineering process. So the architecture is divided in three main parts:

1. **Artifact Independed Layer:** The layer is mainly represented by the client, the Component-Logger and, additionally, the database.
2. **Artifact Depended Layer:** Depending on the type of artifact different steps have to be performed. Those are described by the Artifact-Layer and their corresponding schedulers and agents.
3. **General Database Layer:** It implements the database and naturally the interfaces to manipulate the database.

First, the framework realizes the traditional Client-Server architecture pattern. The client is re-

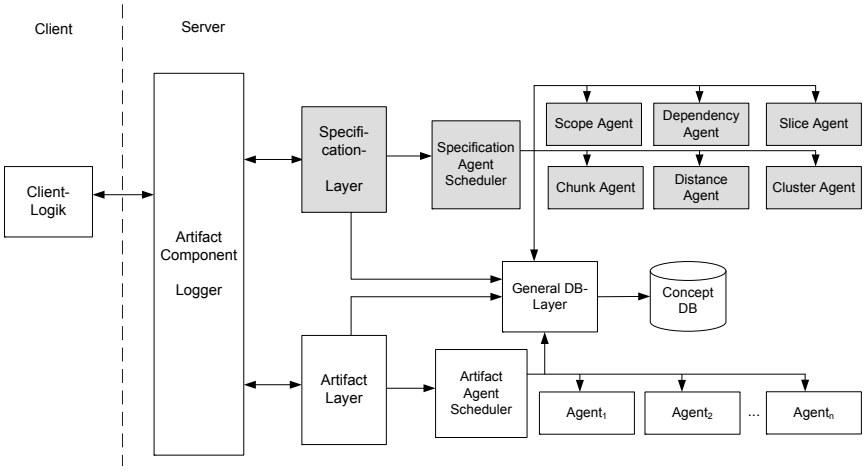

Figure 2: Architecture implemented by the framework.

sponsible for visualizing the results and for triggering the concept extraction. Secondly, the server side is construed to handle different kinds of documents. Currently formal Z specifications are supported. The *Artifact-Component Logger* on the server is responsible for the registration of different *Artifact-Layers*. Depending on the document to be stored or accessed the Logger identifies the right kind of layer for further processing steps. For example, for a Z-Specification the *Specification-Layer* is contacted. Those layers decide based on the current task which functionality should be performed. For difficult tasks it is possible to delegate the work to so called agents. Those *Agents* are activated through the *Artifact Agent Scheduler*.

Whenever a new document is stored in the database, different analysis tasks have to be performed to extract concepts. The analysis/extraction process is done by so called agents. Each agent is responsible to extract one specific kind of concept. So the *Dependency Agent* extracts data and control dependencies. The *Cluster Agent*, for example, is used to calculate all possible clusters within one document. This Agent depends on the *Dependency Agent*, as clusters are created using connected parts. More information about agents is given in Sec. 3.2. All the concept information extracted by agents are stored within a database.

3.1 Database

The database, responsible for storing the identified concepts, is divided into four parts (for an overview see Fig. 1, for details see App. A, Figures 5 and 6). They arose from an evaluation of documents and their possible concepts. The situation was seen as a multi-dimensional problem, and the dimensions where mapped to those parts (Pohl, 2008).

Since it is possible to handle different types of documents, those documents have to be stored accordingly. The *Management/Project Pane* is responsible for this functionality. Therewith, it is possible to store different *Artifacts* of the software engineering process. This is done depending on the related *Project* and *Phase* within they are generated. An artifact consists of different *SyntaxElements*. As mentioned before, in the area of Z those elements are named Primes. They form basic concepts of the artifact and are stored in the *(Syntactic) Representation Pane* of the database. The underlying structure of the document and respectively of the elements can be any simple or complex graph and is expressed by a circular n to m relationship. For further processing steps, like dependency calculation, its necessary to annotate those elements. One sort of annotation could be the usage of identifiers within primes, as mentioned earlier in Sec. 2.2.

Based on this annotation and syntactical elements, concepts are extracted. The identified concepts are handled within the *Model Concept Pane*. There are different types of concepts, e.g. dependencies, slices, cluster and, chunks. Concepts can form hierarchies. To express these relations, an n to m recursive relationship is chosen. For each concept, it is possible to store a *ConceptMetaData* information.

Additionally, different views on documents might exists. Those views cluster concepts of the same type together (e.g. all control dependencies of the document). Views are represented within the *(Concept) View Representation Pane*. The set of all views can be seen as a semantic snapshot of the whole document.

As mentioned earlier, each type of concept is identified by an associated agent, which is also respon-

sible to cluster the concepts in corresponding views. Which agent is activated, depends on the actual event performed by the user.

3.2 Agents

The main part of the architecture is captured by so called agents. Agents are used in the framework to perform complex tasks. They are running independently from the client on a server with the corresponding database information. It is also possible to extend the framework with additional agents in the course of time. The current state of the framework distinguishes between the following different kinds of agents:

- Scope Agent
- Dependency Agent
- Distance and Cluster Agent
- Slice Agent
- Chunk Agent

Since different types of dependencies (data and control dependencies) are necessary to identify other kinds of concepts (e.g. clusters), those have to be extracted from the syntactical structure of the artifact. This extraction process is performed by the *Dependency Agent*.
Dependency calculation naturally depends on scopes (like in programs), which must also be extracted from the syntactic representation. The *Scope Agent* is theoretical responsible for the extraction of scopes. In praxis, this agent is divided into different subagents, for each type of scope one agent. So, the *Dependency Agents* operates on the scopes and on the annotation of syntactical elements. Both, dependencies and scopes, are stored as concepts within the database.
Another agent is the *Cluster Agent*. The task of the agent is to identify all possible clusters of one specific specification document. The agent calculates all potential clusters of one artifact. For clustering Z-specification distances between primes are drawn on. So, another agent (*Distance Agent*) calculates the distance of syntactical elements.
Other very important concepts are slices and chunks. So, two agents (*Slice and Chunk Agent*) are responsible for extracting slices and chunks depending on several parameters and various kinds of dependencies.

Obviously, some of those agents have to wait for others agents to finish. For example, the agent for the control dependencies has to wait for the agents, responsible for creating different scopes. In this framework, everytime a document is stored, scopes, dependencies, and clusters are calculated automatically. If the client needs slices or chunks, the corresponding agents are explicitly invoked.

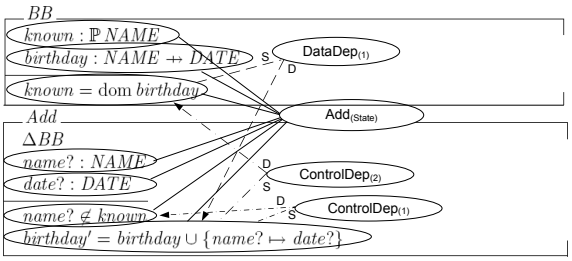

Figure 3: Add-Schema scope and resulting data and control dependency (based on the Birthday Book (Spivey, 1989)).

3.3 Dependency Agent

The *Dependency Agent* is described hereinafter in more detail, as it is used in Sec. 5 to demonstrate the evaluation results. With the identified scopes (also stored as concepts), it is possible to calculate dependencies between syntactical elements.
Within the Dependency Agent, data and control-dependencies are calculated. With the following two scopes it is possible to calculate those dependencies:

- **State Scope:** The state scope deals with schema inclusions within a specification document, and it aggregates the primes of the included scope and the primes of the including section.
- **Connectivity Scope:** The connectivity scope merges all primes of two or more schemata that are combined via schema operations.

An example for the state scope for the Add-Schema of a Z Specification is shown in Figure 3. The $Add_{(State)}$-label describes the state scope of the operation.

To identify control dependencies, some rules have to be followed. A syntactical element is control depend, iff there is another element, which decides if the prime is evaluated or not. Through annotations it is possible to find depending elements.
Here one strength of the framework can be shown: As annotations and scopes are stored within the database, this can be done with small queries:

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

$$\Pi_{sid}(\sigma_{Concept.id=act}Concept \bowtie (\sigma_{ConceptType.name="Inclusion"}ConceptType)) \quad [sid \neq sid]$$

$$\Pi_{sid}(\sigma_{\substack{AnnotationType.name\neq"T" or \\ AnnotationType.name\neq"C or AnnotationType.name\neq"D"}} SyntaxElement \bowtie ElementMetaData \bowtie AnnotationType) \quad (2)$$

The queries calculate the start and the end position of the control dependency arcs. Query (1) identifies the start position of the dependencies. The *act* identifier describes the scope for which the current calculation is performed. The end arcs are calculated via the query (2).

The dependency agent takes all elements resulting from the first query and connects them with the resulting elements of the second query. The same is done for data dependencies. However, it is used with different queries, checking for different annotation types. Additionally, the identified pairs of dependencies are stored as concepts, within the database. For the Add-Operation the dependencies shown in Figure 3 are identified.

Due to the fact that maintenance was an important issue in this framework, the choice to use the EJB[2]-Technology for the realization was made. For this reason the following sections represent a discussion and the findings to use this technology.

4 EJB

EJB 3.0 (Enterprise Java Beans) is a server-side middleware architecture of Sun Microsystems. The reason for choosing this technology was to achieve the non-functional requirement *maintenance* of the framework. Therefor, EJB facilitates the separation between the application and the database logic (SunMicrosystems, 2007). Additionally, it offers Bean-Objects to handle data easily and to map the relational data format to the object oriented paradigm respectively. This fact could be understood as an abstraction of the relational database schema in an object oriented presentation.

4.1 Overview

The EJB technology is implemented via corresponding Java classes on the server which run in the EJB-Container. For the implementation of various functionalities different kinds of beans are provided (SunMicrosystems, 2007):

- **Stateless Session Beans:** It represents a stateless Java class, so the state is only maintained during the method call from the client. Clients could call simultaneously the Stateless Session Bean. The access is done through public interfaces.
- **Stateful Session Beans:** This bean serves only one client and maintains the session state during its calls. After the call the state is not kept any longer. The access is also done through public interfaces.
- **Message Driven Beans:** The Message Driven Bean enables an asynchronous communication between the client and the bean instance. Therefor, the communication is done through message exchange. Of course, with this kind of bean it is possible to serve more than one client.

Those types of beans are needed to implement the business logic. For the connection to the database special beans (Entity Beans) are needed:

- **Entity Beans:** One object of an Entity Bean class holds one row of the appropriate table. Thus beans are the results of the object-oriented mapping provided by this technology.

All mentioned beans run on an application server within an EJB-Container. This container is responsible for various tasks, like transaction management, security issues and access control (SunMicrosystems, 2007). For the deployment of the beans on the application server, a module (*.jar-File*) with different XML configuration files is needed. For additional information to the EJB technology see the specification (SunMicrosystems, 2008) and the tutorial (SunMicrosystems, 2007).

4.2 Realization

The implementation of the concept management framework is realized via *Java 1.6*, *EJB* and the Open-Source database system *MySql*. For the server side implementation the application server *Glassfish*[3] from Sun was used. For the development environment NetBeans IDE 6.0.1 was utilized. The ORM (Object-Rational-Mapping) is provided by the *TopLink* persistence provider, developed by *Oracle*.

The framework implements the architecture described in Sec. 3. The client is a stand-alone remote client and thus not executed in an EJB-Container.

[2] Enterprise Java Beans

[3] For further information about Glassfish see: https://glassfish.dev.java.net/, Last visit: May 2008.

On contrary, the server is implemented via EJB. The interface to the client (*Artifact Component Logger*) is represented by a stateless session bean. The different artifact dependent layers (as the *Specification Layer*) are also implemented as stateless session beans. Thereby, it is possible to serve more than one client at a time. This layer is also responsible to start the *Agent Scheduler* for further, complex processing steps. As mentioned earlier, the class is responsible to start the agents in the right order for processing.
The interface to the database are formed by entity beans, which map the database relations to the object oriented classes. Thus, this beans are contained as part in the *General DB-Layer*.
The *Agent Scheduler* for specifications is a traditional Java class, which consults the agents as needed. The *Agents* are also traditional Java classes. For the persistency of the identified concept they get the current entity manger for this session.
With the manager it is possible to work with and access Entity Beans as needed. For example, it is possible to persist entities with the *persist*-method. The manager is also necessary to set up the specially defined EJB-QL[4] queries. The EJB-Query Language facilitates to set up queries, which results in objects returned from the relational database, of course after intermediate processing steps.

5 EVALUATION

The evaluation of the framework was carried out in two steps. First, the usefulness and correctness of the identified concepts were checked, and the derived clusters, slices, and chunks were checked against well-established results to be found in literature[5]. Secondly, the performance was analyzed. The reason for this step was the inexplicable performance lack when working with specifications of raising sizes. The experiences we gained hereinafter described in more details.

The evaluation was based on wide-spread specifications, known as Birthday Book(Spivey, 1989), Petrol Station (Bollin, 2004), and Elevator (Chang and Richardson, 1994). Additionally, an own specification (called Cinema) (Pohl, 2008) was examined. Table 1 presents the complexity of the specifications by exemplifying the number of control and data dependencies.

[4]EJB-Query Language

[5]The semantic evaluation is not within the scope of this contribution. The interested reader is referred to (Bollin, 2008) for more details on the meaning of specification clusters, slices, and chunks.

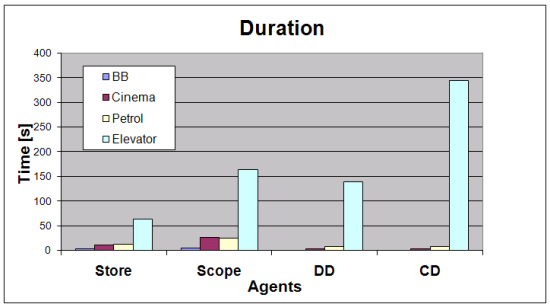

Figure 4: Time needed for storing primes, calculating the scopes, and calculating data and control dependencies.

Specification	CD	DD
BB	10	5
Cinema	121	43
Petrol	192	177
Elevator	1,628	992

Table 1: Complexity, described by the number of data (DD) and control dependencies (CD).

The performance of the system varied, depending on the size of the specification, which was as expected. Complexity considerations showed that the runtime complexity[6] is in $O(cs * 2n_s)$. Figure 4 summarizes the time needed for the identification and storage of all primes and related dependencies. The most complex artifact we evaluated was the *Elevator* specification, and after about 10 minutes the whole artifact was analyzed and stored persistently for later use. This was about 6 times faster than done by former systems (Bollin, 2004). But we were eager to know why it took 5 minutes to store a bit more than 2600 data-sets.

We investigated further into this issue and made two important observations:

- Too much time is lost due to the EJB's synchronization between the database and Java's internal objects.
- There is a very high execution time latency between EJB queries and their corresponding JDBC queries.

The measured times varies due to the different complexities of the specifications. But, performance is lost due to the overhead of the relational and object-oriented mapping. EJB can be seen as an additional layer between the database and the implemented business logic, and every synchronization contributes to an increase in processing time. To get

[6]Here, *cs* is the number of different scopes, and n_s is the number of prime elements.

unique identifiers for objects (our choice was to use *auto_increment ID*), one has to flush/synchronize the objects with those in the database. That this flush is costly was clear, but we wanted to know how much time is lost. So we used the Elevator specification to measure this, and found out that this overhead is about one-third of the total time (see Table 2).

	incl. overhead	no overhead	diff (in %)
DD	139.2	93.3	-32.9
CD	343.6	232.5	-32.3

Table 2: Time (seconds) for concept manifestation including the synchronization overhead and without the overhead.

The second issue we were curious about was the difference between EJB and JDBC when accessing the database. And indeed, we found a big time latency between EJB and JDBC queries. Not surprising, JDBC was faster, but the differences were notable (see Table 3). To measure it, we implemented the same request to the database with the EJB query language and with JDBC statements[7]. Then both requests were issued up to 1000 times[8]. We found out that JDBC scales with the factor of six times better than EJB.

Runs	JDBC [ms]	EJB [ms]
100	188	1,156
1000	1,328	8,343
10000	14,534	99,876

Table 3: Comparison of JDBC and EJB (in milliseconds) access to the database.

So, although EJB (with entity beans and annotations within the entity bean classes) produces a more readable code, performance decreases when one has to store many objects per transaction which are, then, needed in ongoing processing steps. In our framework this is the case when we have to store an object and need the unique ID for storing the intermediate relation between those objects. The flushing/synchronizing mechanism is the only but very expensive way for getting it.

Undoubtedly, EJB yields advantages like transaction management, security mechanisms, and scalability. It offers a comfortable way in implementing things. But this luxury does not come for free.

[7]The query tested for two equal identifiers at different primes and joined two times over the *SyntaxElement*, the *CombinationType* entities, and the *combines* and *emd_annotates_SE* relation. The database contained 2600 entries in the *combines* relation.

[8]Database internal optimizations, like cashes were, of course, disabled.

6 CONCLUSIONS

Concept location is a challenging task, which also holds for the identification of concepts within formal Z specifications. Once detected, they should be stored for future use. For this reason a framework was implemented that is able to identify *and* store concepts in a database. For its implementation the middleware technology EJB was applied.

This paper introduces the general architecture and evaluates the resulting framework. The evaluation shows that it produces correct and useful results. However, the performance of the frameworks is strongly influenced by EJB. We found out that the most important latency is due to the synchronization process between a bean objects and the database.

The comparison between JDBC and EJB shows a high factor of performance loss. JDBC scales six times better than EJB in terms of runtime. Additionally, EJB implements an intermediate layer and, therefore, runs into some performance latencies.

In the case of the framework we finally decided to fully replace the EJB technology by JDBC. The comparison is just ongoing, but due to our experiments with the current system we expect a higher performance of the factors 4 to 5.

REFERENCES

Bollin, A. (2004). *Specification Comprehension Reducing the Complexity of Specifications*. PhD thesis, Institute for Informatics-Systems, University of Klagenfurt.

Bollin, A. (2008). Concept Location in Formal Specifications. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(2):77–104.

Borland (2008). The Rational XDE Homepage. http://www.borland.com/us/products/together.

Burnstein, I., Roberson, K., Saner, F., Mirza, A., and Tubaishat, A. (1997). A Role for Chunking and Fuzzy Reasoning in a Program Comprehension and Debugging Tool. In *TAI-97, 9th International Conference on Tools with Artificial Intelligence*. IEEE press.

Chang, J. and Richardson, D. (1994). Static and Dynamic Specification Slicing. In *In Proceedings of the Fourth Irvine Software Symposium, Irvine, CA*.

Chen, K. and Rajlich, V. (2001). Ripples: Tool for change in legacy software. *Software Maintenance, IEEE International Conference on*, 0:230.

Ebert, J., Kullbach, B., Riediger, V., and Winter, A. (2002). GUPRO – Generic Understanding of Programs An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2).

Eclipse (2008). Generative Modeling Technologies Homepage. http://www.eclipse.org/gmt/.

Ferenc, R., Beszedes, A., Tarkiainen, M., and Gyimothy, T. (2002). Columbus – Reverse Engineering Tool and Schema for C++. In *IEEE International Conference on Software Maintenance*, pages 172–181, Montreal, Canada.

Hoare, C. A. R. (1981). The emperor's old clothes. *Commun. ACM*, 24(2):75–83.

Jouault, F. (2005). Loosely Coupled Traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA 2005), Workshop on Traceability*.

Korshunova, E., Petkovic, M., van den Brand, M. G. J., and Mousavi, M. R. (2006). CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code (Tool Paper). In *Working Conference on Reverse Engineering (WCRE'06)*, Benevento, Italy.

Mittermeir, R., Bollin, A., Pozewaunig, H., and Rauner-Reithmayer, D. (2001). Goal-driven combination of software comprehension approaches for component based development. In *Proc. Symposium on Software Reusability*, ACM Press, pages 95–102.

Müller, H. A., Tilley, S. R., and Wong, K. (1993). Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project. In *CASCON'93*, pages 217–226.

Nickel, U., Niere, J., Wadsack, J., and Zündorf, A. (2000). Roundtrip Engineering with FUJABA. In Ebert, J., Kullbach, B., and Lehner, F., editors, *Proceedings of 2nd Workshop on Software-Reengineering (WSR)*, Bad Honnef, Germany.

Pohl, D. (2008). Specification Comprehension – Konzeptverwaltung am Beispiel zustandsbasierter Spezifikationen (in German). Master's thesis, University of Klagenfurt, Software Engineering and Soft Computing.

Rajlich, V. and Wilde, N. (2002). The Role of Concepts in Program Comprehension. In *International Workshop on Program Comprehension*, pages 271–278. IEEE Computer Society.

Spivey, J. (1989). *The Z Notation. C.A.R. Hoare Series*. Prentice Hall.

SunMicrosystems (2008). Enterprise JavaBeans Specification 3.0. http://java.sun.com/products/ejb/docs.html Page last visited: April 2008.

SunMicrosystems, I. (2007). The Java EE 5 Tutorial. For Sun Java System Application Server 9.1. http://java.sun.com/javaee/5/docs/tutorial/doc/JavaEETutorial.pdf.

Wakounig, D. (2008). *Reverse Engineering of Typed Rule-based Systems – Dependency Analysis and Comprehension Aspects*. PhD thesis, University of Klagenfurt.

APPENDIX

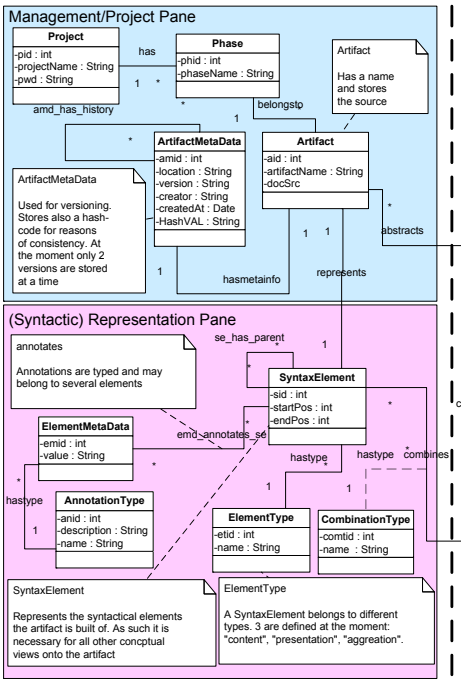

Figure 5: Management and Representation pane of the database model.

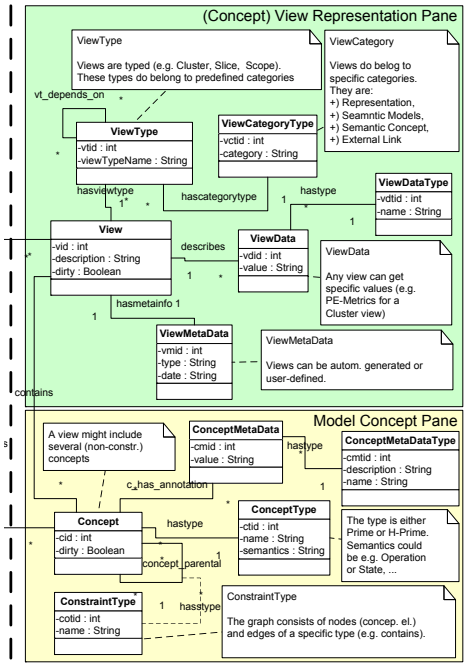

Figure 6: Concept and View pane of the database model.