Using the TGraph Approach for Model Fact Repositories

Daniel Bildhauer, Jürgen Ebert, Volker Riediger, and Hannes Schwarz

Institute for Software Technology, University of Koblenz-Landau
{dbildh, ebert, riediger, hschwarz}@uni-koblenz.de

Abstract. Nowadays models, being abstract representations of software artifacts such as architecture descriptions, test cases, or source code fragments, play an important role in software technology. This paper shows how graph-based repositories can be used to keep these models together with their interconnections. One of the possible, and implemented, applications is the identification and retrieval of artifacts which are potentially reusable in an ongoing development project. The abstract representation retained in a repository allows for employing querying technology to precisely find those artifacts suiting this specific purpose.

Key words: Repository Technology, TGraphs, Metamodeling, Querying, Reuse

1 Introduction

Modeling has become a much-used enabling activity in modern software technology during the last two decades. Model-based approaches are being introduced for almost all activities in software development and software evolution. In these contexts the term **model** is used for the abstract description of relevant parts of something thereby neglecting parts that are irrelevant for context of discourse. A model is always intended to serve some given purpose and contains the information that is **relevant** for this purpose. The usefulness of modeling heavily depends on the concrete **modeling approach**, its formal background and the tool support provided for it.

In a large software development project literally hundreds of **artifacts** have to be produced which model different views on the system in the course of given software development process. There are requirement lists, scenarios, domain glossaries, architecture sketches, class diagrams, source codes, test cases, configuration files and many more, all of which may be viewed as special models of parts of the system or its environment.

Each of these artifacts is written in some specific **artifact language**. Some of these artifact languages are visual, some are textual, many are hybrid. Some artifact languages are formal, others are only formatting. As an example, RSL [1] is a language for defining terminology, specifying requirements and scenarios and is intended to support requirements specification. It supports textual requirements by a restricted natural English language. As another example, UML [2] provides a set of visual languages for defining the structure and behavior of the system especially during coarse and detailed design. Thus, the term language is being used in a very broad sense in this context.

All these artifacts have to be developed and evolved during the course of a software development project. Therefore they have to be stored in a versioned **artifact repository**, which is needed to keep track of all artifacts, their changes and thereby allows tracing the development and reconstructing older versions and system configurations if needed.

In all integrated approaches to software development, be they model-based, component-based, service-based, or product-line-based, the developed artifacts are interconnected by using common elements (terms, classes, definitions) and by inter-artifact links, which are usually called **traceability links**. Traceability information can be a part of the artifacts, but there can also be separate artifacts which keep this information. Configuration files and build scripts are typical examples for artifacts that describe the connection between different artifacts and help to integrate the artifact set.

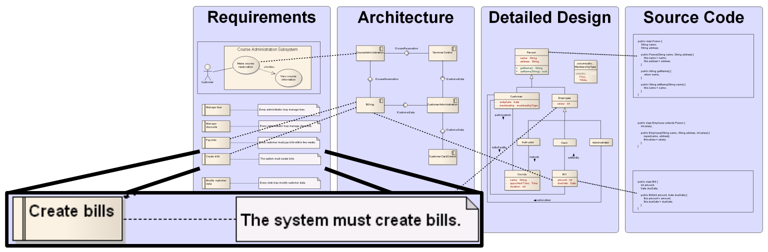


Fig. 1. Interconnected artifacts formulated in RSL, UML and Java. The excerpt shows a textual requirement.

Figure 1 sketches some typical software engineering artifacts depicting models from requirements, architecture, detailed design, and source code of some system including some inter-artifact links which add traceability information.

Storing the artifacts in a versioned artifact repository does not suffice to keep track of all interconnections and dependencies between them, since artifacts are still rather course-grained entities. Here, a more **fine-grained view** on the artifacts is necessary which allows accession of more detailed, but relevant information contained in them. According to a metamodeling approach the views of the artifacts in the artifact repository can be modeled in a corresponding **fact repository**. Using a common unifying metamodeling approach for all artifact languages, a common fact repository can be created which contains the facts of all relevant artifacts in a unified and integrated representation.

In this paper we describe the metamodeling approach used in the **ReDSeeDS project**[1] [3], namely the **TGraph approach**, where the facts extracted

[1] www.redseeds.eu

from the artifact are stored as TGraphs [4], i.e. typed, attributed, and ordered directed graphs. TGraphs are a very general and powerful kind of graphs and provide all needed and desired features for representing facts as graphs in a tool. An efficient implementation of TGraph, the JGraLab Java API [5], enables the easy use of TGraphs in practice.

The goal of the ReDSeeDS project is the creation of an open framework for case-based software reuse. Here, requirements for software projects are described in the semi-formal **Requirements Specification Language** (RSL). All artifacts developed on top of such RSL descriptions for a given project form a so-called **software case**. The artifacts therein are formulated using the **Software Case Language** (SCL) which is essentially UML extended by traceability information and transformation scripts produced in the context of model-driven development. **Reuse** in the ReDSeeDS context will be based on similarity exploration on the basis of requirements and then querying existing software cases for their reusable parts. While software cases are kept in the ReDSeeDS artifact repository all analysis will be done by querying the respective ReDSeeDS fact repository. A task in ReDSeeDS is concerned with the examination of candidate repository technologies. Refer to [6] for a detailed discussion of these technologies together with a comparison of their individual benefits in contrast to the TGraph approach. Technical details and gained insights are described there as well as fields of application for the different approaches ranging from plain text databases to SQL to semantic web technologies such as OWL. A detailed discussion of related work has to be omitted here due to space limitations.

Section 2 introduces the notion of a software case as a set of compatible, integrated artifact versions constituting a software solution, and exemplifies the concept of artifacts and their abstract fact model. **Section 3** introduces the TGraph approach to metamodeling which uses an EMOF-compatible metamodeling language in combination with a graph-based semantics leading to graphs as representations of artifact fact models in a fact repository including their interconnection by traceability links. **Section 4** describes how the fact repository can be explored by queries and how this approach leverages the development process by providing detailed global information to the developer. It also sketches the technical implementation of this approach by using the JGraLab APIs.

2 Repositories

As explained in the introduction the various documents produced during a software development process are referred to as **software artifacts**. Together, these artifacts build up a software case which in its entirety is stored in an **artifact repository**, usually managed by a version management system. In this repository, all artifacts are stored in their native form, e.g. XMI files are used for storing UML diagrams.

To make the information inside the artifacts accessible for analysis, a problem-specific abstraction of the software artifacts is needed. One abstraction that has been proven to be appropriate and flexible are **graphs**, e.g. abstract syntax

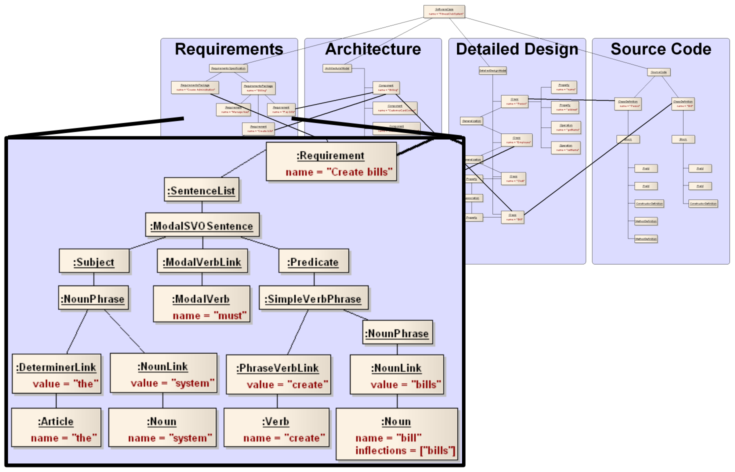


Fig. 2. The abstract syntax graph of a software case. The excerpt shows the subgraph corresponding to the "Create bills" requirement in figure 1.

graphs. Graphs are well-understood, formally defined, efficiently implementable, and easily usable. Due to their high flexibility, graphs offer also the possibility to create abstractions that are adapted to the relevant analysis tasks. In the context of this paper, a complete software case including all its artifacts and all (traceability) links between them is viewed as one single graph, more specifically as a **TGraph**.

Figure 2 sketches a corresponding fact graph for the example shown in Figure 1. Here the natural language requirement named Create bills is stored as an abstract syntax graph.

Analogously to the artifacts themselves, their abstractions as TGraphs are also held in a repository. Since the TGraphs represent facts extracted from the artifacts, this repository is named **fact repository**. It contains all data relevant for searching for reusable partial software cases or for determining the similarity of the requirements parts of software cases.

The relevant facts of the artifacts stored in the artifact repository have to be **extracted** to the fact repository. This fact extraction can be done by special tools - depending on the artifact language used. While facts of textual artifacts such as source code files or even documents in natural language can be extracted using special parsers, an XML representation of, for instance, an UML class diagram may be transformed to TGraphs by an XSLT script. The prerequisite is the existence of an appropriate metamodel defining the structure of the TGraph representation (cf. section 3).

The graph stored in the fact repository is then used as the basis for all relevant analysis tools. To find **similarity** of software cases some graph similarity approach can be directly applied on the data. In ReDSeeDS currently the SiDiff algorithm [7] is used for that purpose. On the basis of the similarity calculated with this algorithm, the reusable partial software cases are identified by **slicing**. The reuse of the cases found in this way is performed by identifying the original artifacts in the artifact repository. These can be opened and edited with the appropriate tools that were initially used to create them.

3 Metamodeling

To store and use the abstractions in the fact repository, the structure of the **abstract syntax graphs** has to be defined. **Metamodeling** offers a basis for defining this relevant information. Given a metamodel for an artifact language, the relevant parts of all artifacts conformant to that language can be extracted as instances of the metamodel. Given a precise metamodel, the relevant information of all artifacts can be extracted to a fact model. Using a common unifying metamodeling approach, all instances are of the same kind and can be handled in a common way. The TGraph approach uses TGraphs as instances of all metamodels.

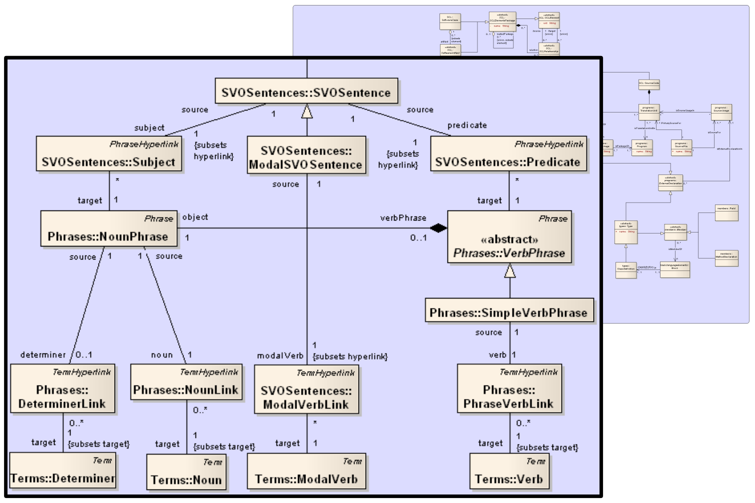


Fig. 3. The SCL metamodel. The excerpt shows that part of the metamodel which the highlighted subgraph in figure 2 is an instance of.

To describe the structure of graphs by metamodels, the metamodels need to have a graph semantics, i.e. they can be interpreted as graphs. Speaking in terms of MOF [8], an EMOF (Essential MOF) compatible metamodel satisfies this constraint: instances of meta-classes represent vertices and instances of meta-associations represent edges. The metamodels for the specification of the fact repository structure in ReDSeeDS may contain all elements of EMOF and a few additional elements from CMOF, such as subsetting and redefinition of rolenames. This subset of UML is called **grUML** (Graph UML). Being a proper subset of UML, grUML is adequate to specify metamodels for graphs.

Figure 3 shows a coarse cutout of such a metamodel for the abstract syntax of a given language. The classes of the metamodel describe sets of vertices of the instance graphs, and the edges of the instance graphs are described by the associations in the metamodel.

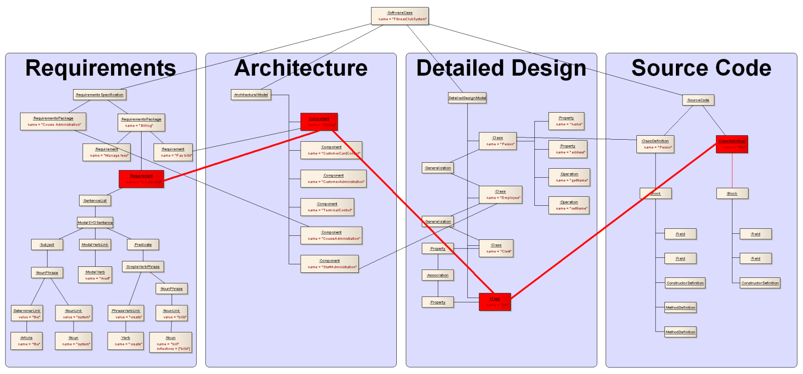
A class that is worth a special consideration is the metaclass **TraceabilityLink** together with its subclasses (not shown here). These are used to model tracebility information between several parts of a whole software case in a fine grained manner. As an example, a requirement is connected to the elements in the architectural and detailed design model that realize it by such links. By traversing these connections, all model elements that are directly or indirectly dependent on the requirement can be identified, thereby slicing the software case.

4 Using the Fact Repository

The graph representation of the artifacts contained in the fact repository can be employed for a variety of applications. Among them are reverse engineering, similarity and difference computation, and traceability. In ReDSeeDS *JGraLab* [5] is used as the implementation basis for the fact repository. JGraLab is a Java class library which features the creation, manipulation, and traversal of TGraphs based on the given metamodels, called schemas in the context of JGraLab. Using JGraLab all relevant information contained in the fact repository is easily accessible by using the **JGraLab API** and/or the **GReQL query facility**. These features are further augmented by an efficient I/O mechanism providing persistence of graphs and schemas.

Querying. An efficient and powerful means of accessing and extracting information from TGraphs is the use of **querying** as an enabling technology [9]. Since the fact repository has a precise metamodel, querying allows for concisely describing requests for specific pieces of information. Since queries are typically short and straight-forward, users are relieved of devising, for example, complex traversal algorithms on the repository.

In ReDSeeDS, employing a fact repository accompanied by a supporting querying technology aids in achieving the project's main goal: to foster the reuse of parts of software cases representing past, finished projects. Concerning such a "past software case", the idea is to identify requirements within it which are similar to requirements in the software case currently under development. Based on the set of identified requirements, the so-called **slicing criterion**, the prospective ReDSeeDS tool will then formulate a query which yields a **slice** of the particular past software case. This slice is computed by utilizing the traceability information interconnecting the representations of the various artifacts within the fact repository. In the result, the slice consists of those artifacts within the software case which are related to the slicing criterion according to the traceability view manifested by the query.



```
elementsIn(
  from req:V{Requirement}, archElem:V{UMLElement},
       desElem:V{UMLElement}, class:V{ClassDefinition}
  with req.name="Create bills" and
       req <−−{Satisfies} archElem and
       archElem <−−{Realize} desElem
       desElem <−−{Implements} class
  report req, archElem, desElem, class
  end
)
```

Fig. 4. Sample GReQL query with associated slice of a software case

Querying in the context of fact repositories based on TGraphs is facilitated by the **Graph Repository Query Language** (**GReQL**) [10]. It features the declarative formulation of queries based on trivalent logics and regular path expressions. GReQL queries have a simple from–with–report–end structure. In the from-part of the query variables are bound to some domains (here: vertex types). The variables can then be used in the with-part to impose constraints on their values. The report-part finally defines the structure of the query's result.

In its upper part, figure 4 shows a slice of a software case computed on the basis of an exemplary GReQL query displayed at the bottom of the figure. The highlighted requirement "Create bills" acts as the slicing criterion. By following the reverse Satisfies links, it is possible to reach vertices representing UMLElements of the case's architecture part. Reverse Realization links constitute the interconnection between UMLElements of the architecture and detailed design parts. Finally, traversing reverse Implements links provides the ClassDefinitions of the source code part which implement the associated design element. For the sake of comprehensibility, the sample query does not take into account *vertical* traceability information, i.e. links between elements within the same part of a software case, such as a Usage link between two UML Components of the architecture. In general, GReQL queries may contain arbitrary regular path expressions and may be nested. Experience in another project [10] proved that GReQL is an effective and efficient means for querying TGraphs. The elementsIn function used in the query yields the set of all Requirements, UMLElements, and ClassDefinitions for which the constraints given in the with part holds.

Using the APIs. Alternatively, the fact repository can also be accessed using the JGraLab API. Going into more detail, JGraLab offers two distinct means of working with TGraphs. The first one is a **generic API** which treats the different types of graph elements as instances of a particular element metaclass. This way all kinds of graph algorithms can be developed on top of the fact graph.

Besides this more "traditional" way of accessing TGraphs, JGraLab also allows for handling the graph elements as pure Java objects. In order to facilitate this, a Java class is generated for each vertex and edge metaclass contained in a schema, thus leading to a purely **object-oriented API**.

```
GraphClass scClass = schema.getGraphClass("SoftwareCase");
Graph sc = new Graph(scClass);
VertexClass reqClass = sc.getVertexClass("Requirement");
req = sc.createVertex(reqClass);
req.setAttribute("name", "Create␣bills");
```

```
SoftwareCase sc = SoftwareCase.create();
req = softwareCase.createRequirement();
req.setName("Create␣bills");
```

Fig. 5. Creation of a requirement using the generic (top) and the object-oriented APIs (bottom).

Using the creation of the "Create bills" requirement as an example, Figure 5 shows the differences between the generic and the object-oriented APIs. Note that for the sake of brevity, some of the displayed method signatures are shortened.

Among a couple of potential fact repository technologies, JGraLab emerged as the fact repository technology of choice in the ReDSeeDS project. Its integrated querying capabilities inherently support the computation of software case slices, and it is able to handle graphs containing millions of elements, a number easily imaginable when dealing with models of real software cases. Furthermore, it proved to be a suitable basis for performing the model transformations envisioned by ReDSeeDS [**?**].

5 Conclusion

This paper described the coordination between the artifact repository and the fact repository for storing models in the context of the ReDSeeDS project. The **fact repository** holds the abstract representations of the software artifacts, corresponding to a defined **metamodel**. Using the TGraph approach, the metamodel corresponds to the fact repository schema and directly supports its implementation by graphs. Then, **querying** can be applied to analyze the artifacts. The use of querying for slicing was shortly sketched. Furthermore the repository is accessible via two APIs.

Future work will be to refine the TGraph approach by incorporating explicit representation of distributed and hierarchical graph structures. Provided these changes are implemented in JGraLab and in GReQL, even more sophisticated artifact representations could be devised, e.g. allowing for an improved flexibility concerning the granularity of traceability relationships. Another yet unexplored issue is the power of graph-based querying compared to logic-based querying concepts. On the basis of such a comparison, a hybrid solution could be developed combining the advantages of both approaches.

References

1. Smialek, M., Bojarski, J., Nowakowski, W., Ambroziewicz, A., Straszak, T.: Introducing a unified requirements specification language, nakom,. In: Proc. CEE-SET Software Engineering Techniques Conference, Software Engineering in Progress. (2007) 172–183
2. Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. (2007)
3. Smialek, M.: Towards a requirements driven software development system. In: MoDELS Conference, Genova, Italy, 2006 (electronic material)
4. Ebert, J., Franzke, A.: A declarative approach to graph based modeling. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: Graphtheoretic Concepts in Computer Science, Berlin, Springer, LNCS 903 (1995) 38–50
5. Kahle, S.: JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen. Diplomarbeit, Universität Koblenz-Landau (2006)
6. Bildhauer, D., Ebert, J., Riediger, V., Krebs, T., Nick, M., Schwarz, H., Kalnins, A., Kalnina, E., Nick, M., Schneickert, S., Celms, E., Wolter, K., Ambroziewicz, A., Bojarski, J.: Repository selection report. Project Deliverable D4.4, ReDSeeDS Project (2007) www.redseeds.eu.
7. Kelter, U., Wehren, J., Niere, J.: A generic difference algorithm for UML models. In: Proceedings of the SE 2005, Essen, Germany, Essen, Germany (March 2005)
8. OMG: Meta object facility core specification version 2.0. Technical report (2006)
9. Kullbach, B., Winter, A.: Querying as an Enabling Technology in Software Reengineering. In Verhoef, C., Nesi, P., eds.: Proceedings of the 3rd Euromicro Conference on Software Maintenance & Reengineering, Los Alamitos, IEEE Computer Society (1999) 42–50
10. Ebert, J., Kullbach, B., Riediger, V., Winter, A.: GUPRO. Generic Understanding of Programs - An Overview. Electronic Notes in Theoretical Computer Science **72**(2) (2002)