Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Graph-based Traceability – A Comprehensive Approach

Hannes Schwarz[1], **Jürgen Ebert**[1], **Andreas Winter**[2]

[1] Institute for Software Technology, University of Koblenz-Landau, Koblenz, Germany
[2] Department for Computer Science, Carl von Ossietzky University, Oldenburg, Germany

The date of receipt and acceptance will be inserted by the editor

Abstract In recent years, traceability has been globally accepted as being a key success factor of software development projects. However, the multitude of different, poorly integrated taxonomies, approaches and technologies impedes the application of traceability techniques in practice.

This paper presents a comprehensive view on traceability, pertaining to the whole software development process. Based on the state of the art, the field is structured according to six specific activities related to traceability: definition, recording, identification, maintenance, retrieval, and utilization. Using graph technology, a comprehensive and seamless approach for supporting these activities is derived, combining them in one single conceptual framework. This approach supports the definition of metamodels for traceability information, recording of traceability information in graph-based repositories, identification and maintenance of traceability relationships using transformations, as well as retrieval and utilization of traceability information using a graph query language.

The approach presented here is applied in the context of the ReDSeeDS project that aims at requirements-based software reuse. ReDSeeDS makes use of traceability information to determine potentially reusable architectures, design, or code artifacts based on a given set of reusable requirements. The project provides case studies from different domains for the validation of the approach.

Key words traceability – graph technology – model transformations – software engineering

1 Introduction

Artifacts in software development range from collections of requirement statements over architecture and design models to source code and test cases. All these artifacts are strongly interdependent. In order to keep all relevant documents in a consistent but still interconnected state, their interdependencies have to be managed appropriately to determine the impact of change for concrete software elements. This necessity has led to the notion of *traceability management*.

The need for an appropriate treatment of traceability information has even increased with the advent of model-driven development methods. Focusing on models and model transformations leads to more explicit knowledge on the traceability connections between different software elements.

This paper starts by categorizing the various aspects of traceability into specific *activities* based on the state of the art, ranging from the definition of a metamodel for traceability information to the retrieval and maintenance of traceability information. By offering this categorization for the various aspects within the field of research, the foundation for a common taxonomy is provided.

Existing traceability approaches often deal with isolated aspects belonging to one or only a few activities. Consequently, they neglect the whole picture, i.e. how the approaches interrelate with each other. Therefore this paper promotes a *comprehensive view* on traceability. The comprehensiveness has to be interpreted in two senses. Firstly, the approach incorporates all the aforementioned traceability-related activities. Secondly, it is not limited to specific kinds of artifacts, for example requirements or the source code, but it is supposed to encompass all artifacts created or used in the course of software development projects.

The approach presented in this paper supplies a *seamless* combination of all activities related with traceability information in one single conceptual framework including an implementation. All relevant traceability-related activities (defining, recording, identifying, maintaining, retrieving, and utilizing traceability information) are supported by a coherent *graph-based implementation technology* leading to a smooth integration of these activities in a complete methodological approach.

By using graph technology, various challenges posed by traceability can be consolidated to a joint technological foundation. This basis is simultaneously precise and efficient and allows for the application of a great body of knowledge on graph algorithms and graph analysis. The implementation is based on the *TGraph* approach to graph-based modeling [23].

The approach described here has been applied in the *ReDSeeDS*[1] project aiming at the development of a *Requirements Driven Software Development System* [56]. More precisely, ReDSeeDS aims at reusing software engineering artifacts by comparing the requirements specification of a software system to the requirements specifications of already finished development projects. Then, based on identified similar requirements, it is feasible to follow traceability relationships to find other artifacts realizing these requirements, being potential candidates for reuse in the current project. ReDSeeDS features a model-driven approach for transforming requirements specifications to architecture and detailed design models and further on to source code fragments. The overall approach is empirically validated on the basis of real-life projects provided by the industrial partners in ReDSeeDS, stemming from the domains of banking, supply chain management, and geo-information systems.

The paper starts with a general discussion of the term traceability and its related activities in section 2, including a short survey of related work. Section 3 introduces the TGraph approach to graph technology. The following sections discuss the relevant activities: section 4 deals with definition and recording of traceability information using a graph-based repository technology, section 5 describes the identification and maintenance of this information using model transformations, and section 6 presents retrieval and utilization employing a graph query language. All sections have the same basic structure: Firstly, relevant concepts of the respective activities are introduced. Then, the implementation of these concepts on the basis of graph technology is described. Finally, their concrete application in ReDSeeDS is shown. Section 7 concludes the paper.

2 Traceability

Looking at the existing body of literature, the origins of traceability as a field of research lie in requirements management [1]. Many authors dealing with traceability focus on a system's requirements specification as the center of their research activities, resulting in the term *requirements traceability* which has been used extensively [27,53]. In recent years, the research community started to devote more effort to the exploration of traceability of other artifacts in a software development project, such as source code, design, and documentation [4,58,62]. In [1, 5] traceability is understood as a comprehensive concept encompassing the whole development process, without putting special emphasis on the requirements specification.

The following section 2.1 introduces the notion of traceability by discussing various definitions. Then, in section 2.2, different traceability-related activities are described, each one covering different aspects of this field of research. Categorized according to these activities, a short survey of related work is given. Finally, section 2.3 formulates the *traceability challenge* addressed by the approach presented in this paper.

2.1 Definitions of Traceability

One of the earliest definitions of traceability has its origins in a work of Gotel and Finkelstein [27]:

> "Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction [...]"

It can easily be noticed that this definition has been formulated in the context of *requirements traceability*, severely limiting its usefulness for the comprehensive traceability approach which is promoted in this paper.

In [1], Aizenbud-Reshef et al. coin the term of *model traceability*. They use it to subsume various aspects of traceability in the context of model-driven development, especially concerning metamodeling and model transformations. Although the ReDSeeDS project used as application scenario in this paper is situated in a model-driven context, the presented concepts are expected to be applicable to other methodologies as well. Thus, similarly to requirements traceability, the scope of model traceability is perceived as being too narrow for the purposes of the approach presented.

A more general definition can be found in the IEEE's Standard Glossary of Software Engineering Terminology [32]:

> "The degree to which a relationship can be established between two or more products of the development process [...]"

Spanoudakis and Zisman offer another definition considering traceability in the scope of the whole development process, additionally including stakeholders and demanding for rationales [58]:

> "[Software traceability] is the ability to relate artefacts created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other, the stakeholders that have contributed to the creation of the artefacts, and the rationale that explains the form of the artefacts."

The definitions supplied by [32] and [58] are both sufficient to cover traceability in the varieties and forms

[1] `http://www.redseeds.eu`

treated in the following, consequently an own definition is not needed. However, in order to avoid the more coarse-granular connotation of *artifact*, intuitively referring to a document or entire model but disregarding fragments such as paragraphs or single model elements, the more general term *traceable entity* shall be used.

An important categorization of traceability used in this paper is the distinction between *intra-level* and *inter-level* traceability [42], referring to the *abstraction levels* of traceable entities. "Usual" abstraction levels are a software system's requirements specification, its architecture, its detailed design, and its source code. While intra-level traceability applies to traceability relationships between entities on the same abstraction level, inter-level traceability refers to the relationships between entities on different levels.

To a certain degree, the distinction of abstraction levels depends on the specific case or even the observer. For example, a requirements specification could be split into a set of user requirements, which are more abstract and more specific than system requirements.

2.2 Traceability-related Activities

In order to structure the field of research and to provide the basis for a common taxonomy, different aspects of traceability can be classified into six *activities*, taken and adapted from [50,59]: *defining*, *recording*, *identifying*, *maintaining*, *retrieving*, and *utilizing* traceability information, i.e. traceability relationships together with traceable entities. In the following, these activities are discussed in more detail, including related work. An overview of the discussed publications is given in figure 1.

Definition	[2] [16] [18] [24] [26] [44] [47] [48] **[52]** **[53]** [62]	Recording	**[9]** [34] [44] [52] [53] [55] [62]
Identification	[1] [4] [14] **[15]** [18] [28] [31] **[33]** [52] [55]	Maintenance	[13] [18] [43] **[59]**
Retrieval	**[41]** **[51]** [52] **[54]** [62]	Utilization	[45] [51] [52]

Fig. 1 Overview of related work. Approaches directly related to the ones presented in this paper are highlighted.

Definition of traceability information refers to the determination of entity types to be traced, and traceability relationship types between them. Existing literature on this activity can be roughly categorized into two groups.

Publications belonging to the first group are concerned with approaches for designing *metamodels* for traceability. One possibility are *reference metamodels* providing a predefined, yet adaptable set of entity and interconnecting relationship types [52,53]. However, it has to be noted that these metamodels focus on requirements traceability, thus neglecting traceability of other entities such as architecture components or source code fragments. Other approaches include tracing variability in requirements and architecture in the context of product line engineering [47] and the usage of ontologies for traceability between source code and documentation [62].

In [24] a so-called *traceability meta-type* specifies the required or recommended properties a concrete traceability relationship type should possess. This generic concept allows for the definition of customized, application-specific relationship types.

Drivalos et al. propose a dedicated *Traceability Metamodelling Language* (*TML*) for defining traceability metamodels [18]. TML features a set of specific metaclass attributes which allow to reflect often recurring patterns in traceability metamodels, e.g. that it is mandatory for entities of a particular type to have an incident relationship of a specific type. By aiding in the generation of transformations between different metamodels, the TML is also expected to ease integration of tools with different traceability metamodels.

Although the above-mentioned approaches distinguish between different types of relationships, the meaning of these types is given only informally in most cases, either by a description in *natural language* or only implied by the type's name.

Addressing this issue, the second group of publications related to the definition activity deals with the types, forms, or meaning of traceability relationships. In [18,52], traceability relationships are augmented by integrity constraints. A further step towards formalization is taken in [16], where relationship types between requirements are mapped to logical and mathematical expressions. In [2], relationship semantics is formalized by *event-condition-action* rules. The specification of relationship semantics by *logic formulae* is proposed in [26].

A shortcoming of many traceability approaches is the fact that they treat traceability relationships as being *binary* only, i.e. as a connection between two entities only. By allowing for *n-ary relationships*, UML [48] and the approach pursued in [44] are two of the few exceptions here.

Recording is concerned with physically holding traceability relationships in form of data structures. There exist two basic variants: On the one hand, relationships can be recorded *within traceable entities* by introducing textual references or hyperlinks [34]. On the other hand, numerous concepts for holding traceability information *externally* have been developed, based on different technologies.

The most primitive form of external recording is the usage of a spreadsheet for creating traceability matrices, where rows and columns correspond to traceable entities and cell entries represent traceability relationships between respective entities. This approach is, for instance, chosen by the well-known requirements management tool *RequisitePro*[2]. More sophisticated approaches include the usage of *relational databases* as employed by many traceability tools [53], *deductive database systems* [52], and *open hypermedia* [55]. Latest research investigates the applicability of *graph-based repositories* [9], *XML* and related technologies [44], and *ontologies* [62].

All listed approaches for external recording require a definition of the structure of traceability information to be stored, e.g. by database schemas or metamodels.

Identification is the activity of discovering previously unknown traceability relationships between entities, i.e. the discovery of instances of the relationship types determined by the definition activity. In principle, it is possible to distinguish between *manual* and *automatic* identification. Purely manual identification of relationships is expensive and susceptible to mistakes [1]. Although automatic approaches still do not achieve high levels of precision and recall [58], they relieve developers from the time-consuming burden of manual relationship creation.

Some semi-automatic approaches still demand for manual identification, though to a lesser extent. They automatically infer new relationships based on the ones created by the developers [55] or, on basis of violated integrity constraints, point users to missing relationships [18].

Most authors working on fully automated identification rely on *information retrieval* methods for text comparison [4,31] or they count on *model transformations*, interrelating source and target elements of transformations [15,33]. Obviously, these techniques are applicable for particular entities only. While information retrieval requires text-intensive entities such as requirements documents or user documentation, transformation-based techniques necessitate a model-driven development approach.

Another possibility is to provide guidance on how to create correct relationships by *integrating recording of traceability information into the development process* [52]. More precisely, when developers manipulate entities, a system based on this approach establishes suitable relationships based on the specific steps in the process model they just performed.

More recent approaches comprise the identification of traceability relationships based on *rules* [14] or *runtime analysis and machine learning* [28].

Maintenance is the activity of updating and modifying already existing traceability relationships. Since traceable entities are subject to constant change during system lifetime, traceability relationships have to be adapted accordingly in order to accommodate for these changes. Thus, they are prevented from deteriorating.

Regarding maintenance, similar techniques as for the identification of traceability relationships can be applied [59]. A concept based on the *event-action paradigm* specifically addressing maintenance can be found in [13]. It requires traceable entities to be registered at a so-called event server which monitors them for changes and subsequently adapts incident relationships as needed. A tool based on the same paradigm is *traceMaintainer* [43], using rules in an XML-based syntax to specify the actions to be performed.

Traceability metamodels defined using the TML [18] feature special attributes ensuring the adherence of relationships to certain constraints. Violation of these constraints can be reported to the users in order to provoke a proper reaction.

Retrieval addresses finding and gathering of traceability information which is relevant to specific applications, provided it has already been identified and recorded. Subsequently the retrieved information is delivered to the user.

Existing literature on this activity almost exclusively deals with suitable technologies for retrieval, depending on the employed recording approach. Examples are *SQL* for relational databases, the *Graph Repository Query Language* (*GReQL*) [41,54] for graph-based repositories, and ontology query languages such as *nRQL* [62] for ontologies.

The approach in [51] makes use of regular expressions to describe patterns of traceable entities and interconnecting relationships to be retrieved. In [52], a suite of tools is presented which allows users to retrieve traceability information by textually specifying certain search criteria as well as by graphically highlighting related entities when any one traced entity is selected in a browser.

Utilization deals with using previously retrieved traceability information in concrete application scenarios. This activity takes a somewhat special role among the six traceability activities. Although being an important step in the "lifecycle" of traceability information, this is beyond research on traceability in the narrower sense. For this reason and due to the multitude of different applications, a discussion of related work is out of scope in this context.

However, traceability information can also be presented to the user directly, demanding for means of *visualizing* traceability information. Besides the already mentioned traceability *matrices* and other tabular, text-based representations, e.g. as promoted in [52], a further "natural" form of illustrating the structure of traced entities and their interconnecting relationships is a *graph* [51]. The prototypical tool presented in [45] makes use of

[2] http://www-01.ibm.com/software/awdtools/reqpro/

a modern-style user interface for visualization and navigation of traceability information.

2.3 A Challenge for Traceability: A Comprehensive Approach

The short literature survey given in section 2.2 introduces many different techniques, methods, and approaches dealing with traceability. However, according to figure 1, where only reference [52] covers five activities and [18,62] comprise three activities, most traceability approaches are concerned with one or two specific aspects of traceability only. Examples for such aspects are the definition of a reference metamodel for tracing requirements, the identification of traceability relationships between code and documentation, or the usage of particular query languages for retrieving traceability information.

These approaches are developed in a clearly defined and often optimized embedding, using the most appropriate implementation techniques. But they are also defined rather isolated, using different, not necessarily combinable techniques. This consequently hampers their integration into a comprehensive traceability environment. Current research on traceability is lacking a consistent approach encompassing all traceability activities.

Consequently, a pressing challenge of traceability research is the development of a *comprehensive* approach, i.e. an approach supporting all traceability-related activities from defining, recording, and identifying, to maintaining, retrieving, and utilizing traceability information. Furthermore, the traceability approach has to be *seamless*, meaning that a consistent conceptual and technological foundation for definition and recording of traceability information is existing, spanning all abstraction levels of a software system. This enables the integration and subsequent smooth interaction of different identification and maintenance techniques. Finally, uniform retrieval and utilization of traceability information is facilitated, again regardless of the types of the involved entities and relationships.

As explained in the remainder of this paper, *graph technology* provides such an integrated and consistent approach to tackle this challenge. Based on a formal foundation, the presented TGraph-approach allows to define the structure of traceable entity types and relationship types as a metamodel. Recording of traceability information is facilitated by an efficient API for creating, manipulating, and traversing graphs conforming to specified metamodels, effectively constituting the basis for a graph-based repository technology.

Integration effort done in ReDSeeDS showed that existing model transformation approaches can be adapted to work on TGraphs, consequently providing the means to identify and maintain traceability relationships using transformations. However, it is thinkable to adapt other identification and maintenance approaches to work on graph-based representations of traceability information. Efficient and expressive means for retrieving and utilizing traceability information is supplied by a query language specifically designed for TGraphs.

Existing approaches which constitute a basis for or are similar to the concepts presented in this paper are highlighted in figure 1.

3 Graph Technology

Seamless support for traceability requires an *integrated approach*, supplying assistance for all the traceability-related activities identified in section 2. It has to allow formal definition, algorithmic identification, persistent recording, query-based retrieval, support for interoperable utilization, and transformation and evolution during maintenance of traceability relationships.

This paper claims that graphs are a versatile means for all cited purposes, since they are equally well suited for formal reasoning about software engineering artifacts as well as for efficient implementation of software engineering tools. In this paper, the *TGraph approach* [23] to graph-based modeling is proposed as the basis for a seamless integrated support of the cited activities.

Among other applications TGraphs proved to be a powerful means for defining models for visual languages [61] and for the implementation of *repositories* in software engineering tools [22]. They are applied in the ReDSeeDS project as well.

Section 3.1 introduces the concept of a TGraph, explains its foundation based on an example and shortly discusses other comparable approaches. In section 3.2, the corresponding metamodeling approach grUML is introduced. Section 3.3 gives a sketch of the transformation language MOLA applicable on top of a TGraph-based repository. Finally, section 3.4 introduces the GReQL query language for extracting data from TGraphs.

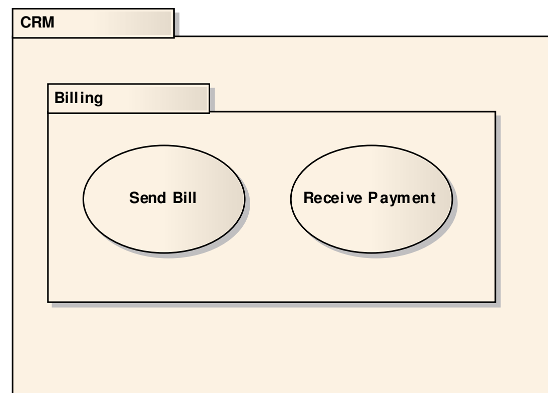

Fig. 2 Use cases of billing subsystem

As a small running example of a software engineering artifact being represented as a graph, a minimalistic *use case diagram* is used in the following. It introduces two use cases of a billing subsystem in a customer relationship management system: Send Bill and Receive Payment, respectively. In order to group the use cases with respect to the addressed (sub)system, RequirementsPackages are used (cf. figure 2).

3.1 TGraphs

TGraphs are directed graphs which are typed, attributed, and ordered. Since all elements, i.e. vertices and edges are typed, appropriate light-weight conceptual modeling is directly supported. Depending on their type, graph elements may carry attribute-value pairs which may be used for modeling data associated to them. The combination of types and attributes leads to an object-based style of modeling. Ordering of the vertices, the edges, and the incidences between vertices and edges also allows for a direct modeling of sequences of entities. A formal definition of TGraphs and their properties can be found in [19].

In TGraphs, edges are first class citizens, i.e. they have all properties such as type and attributes, analogously to vertices. They may be stored in variables and their traversal is supported in both directions.

JGraLab[3] is an efficient API for creating, manipulating, and traversing TGraphs. In addition, there are several supporting technologies, especially concerning schema support (cf. section 3.2) and querying (cf. section 3.4).

TGraphs may be used as a conceptual foundation for repositories which store the *abstract structure* of software engineering artifacts and their traceable entities. Usually all relevant entities are modeled by vertices, and occurrences of entities in some context are modeled by edges. Sequences of occurrences are expressed by the order of incidences.

Figure 3 gives an example of a TGraph, sketched in the style of a UML object diagram. It contains four vertices and three edges forming a graph which can be viewed as an abstract syntax graph (ASG) of the sample use case diagram in figure 2. For illustrative purposes, the order of vertices and edges is represented by their identifiers: v1, v2, ... and e1, e2, ..., respectively. The order of incidences of edges and vertices is displayed by the numbers in parentheses: e1 is the first edge associated to v2 and e3 is the last one.

Important approaches comparable to TGraphs are provided by the libraries *LEDA* [46] and *GRAS* [39] together with its successor *DRAGOS* [11]. LEDA, short for *Library of Efficient Data Types and Algorithms* offers directed and ordered graphs but does not allow for typed and attributes vertices and edges. The kind of graphs

[3] http://jgralab.uni-koblenz.de

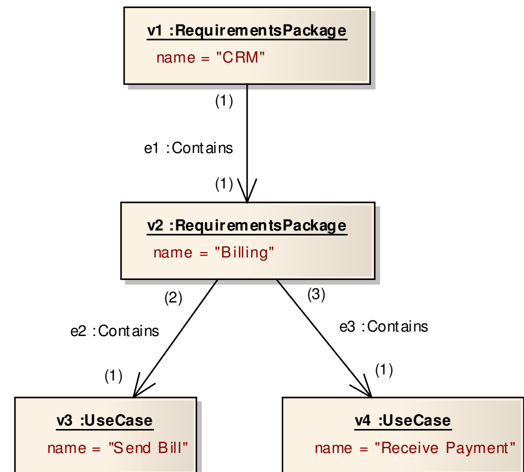

Fig. 3 ASG for the use case diagram in figure 2

supported by GRAS features typed and attributed vertices and typed and directed edges but lacks ordering of edges as well as the possibility to define edge attributes. DRAGOS extends the capabilities of GRAS by permitting types and attributes for edges. Furthermore, while DRAGOS still does not support edge ordering, it can represent hyperedges, i.e. edges connecting edges, and offers a rudimentary approach for hierarchical graphs by nesting graphs within each other. Graphs in GRAS and DRAGOS conform to a defined schema.

3.2 Metamodeling TGraphs – grUML

Sets of TGraphs are specified by a subset of UML class diagrams which actually constitute TGraph schemas. This UML sublanguage is called *grUML* (graph UML) and contains all elements of UML class diagrams that are compliant with a graph-like semantics: classes represent vertex types and associations stand for edge types. To avoid edges connecting edges, association classes must not be connected by associations. The attributes of types are noted in UML style, too, where the notation of associated classes is used to specify edge attributes. Generalization/specialization is used for vertex and edge classes, and multiple generalization is explicitly allowed. Finally, multiplicities are used to denote degree restrictions. The modeling power of grUML is slightly higher than that of EMOF [23]. More precisely, in grUML, it is possible to create models whose expressiveness equals EMOF models, plus some additional features such as edge attributes and sequences of incidences. The latter means that for each vertex, an ordering of its incident edges is maintained.

A TGraph *conforms* to its schema, i.e. it constitutes an instance of the schema, if its element types and attribute assignments as well as the incidences of the edges respect the corresponding descriptions in the schema and if all vertex degrees conform to the multiplicities.

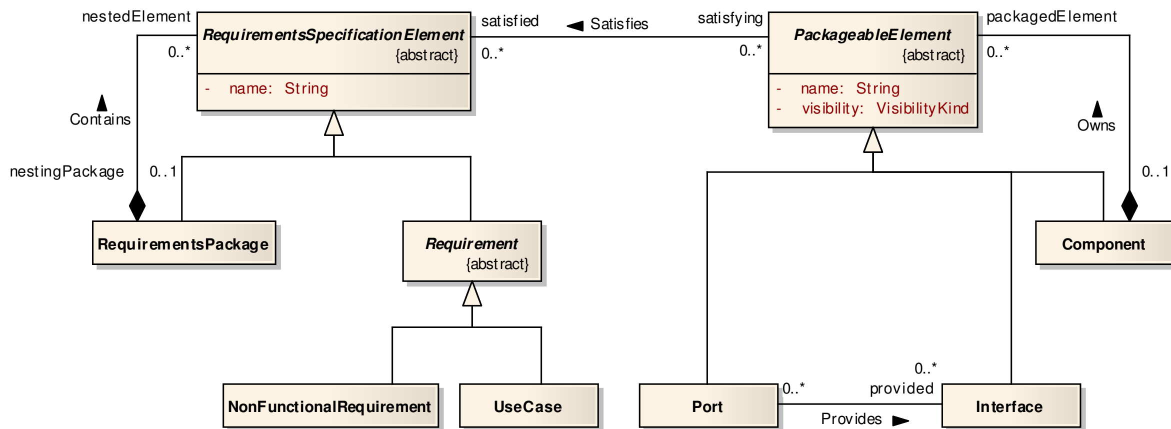

Fig. 4 Schema for the ASGs in figures 3 and 6

All these conditions must respect inheritance. The semantics of inheritance in grUML is similar to standard UML: Firstly, subtypes inherit the attributes of their supertypes. Secondly, instances of edge types may connect instances of subtypes of their incident vertex types.

The notion of a TGraph schema is explained by means of the sample schema in figure 4. The left part of this schema constitutes a suitable metamodel for the ASG in figure 3. In addition, it contains an excerpt of the UML metamodel along with the Satisfies edge class, applicable for modeling and tracing architecture elements generated from the use cases (cf. section 3.3).

3.3 Transforming TGraphs – MOLA

Transformations of software engineering artifacts can be defined on the basis of their abstract syntax, i.e. their TGraphs. A transformation system that is compatible with TGraph modeling and JGraLab is the MOLA-Tool which has been developed by the University of Latvia [35].

MOLA is a programmable graph transformation language, which builds on rules consisting of a pattern and some actions and adds control structures – mainly *loops* – to control the execution of these rules. A MOLA transformation procedure connects a start node with an end node via one or more MOLA rules.

Figure 5 illustrates an example of a MOLA transformation procedure that takes the ASG of figure 3 and extends it by adding corresponding architectural Components, Ports, Interfaces, respectively – leading to the graph depicted in figure 6. That figure also shows Satisfies edges automatically generated by the transformation, serving as traceability relationships connecting architecture elements to their origins in the requirements model.

The example transformation contains three loops (large bold rectangles) each of which iterates a rule. Rules are expressed by gray rectangles with rounded corners. Each rule has one small bold rectangle which denotes a graph vertex and some light black edges and vertices which together specify the context of the particular vertex. The loop iterates over all instances in the graph that match the bold vertex and its context. For each instance the dashed vertices and edges are added to the graph. Note that MOLA only employs role names for denoting edges.

Thus, the example consists of the following three steps:

Step 1: For each RequirementsPackage *rp1*, a satisfying Component *c1* is created. *c1* is connected to *rp1* by a Satisfies edge.
Step 2: For each UseCase *uc2* nested in a RequirementsPackage *rp2* handled in step 1 (i.e. a corresponding Component *c2* was generated), a satisfying Interface *i2* and a Port *p2* is created. *i2* is connected to *uc2* by a Satisfies edge, a Provides edge links *p2* to *i2*, and an Owns edge relates *c2* to *p2*.
Step 3: For each RequirementsPackage *rp3-1* nested in another RequirementsPackage *rp3-2* which are both connected to satisfying Components *c3-1* and *c3-2*, respectively, an Owns edge packaging *c3-1* in *c3-2* is generated.

Applying this MOLA procedure to the graph presented in figure 3 results in the graph shown in figure 6, which itself complies to the graph schema given in figure 4.

3.4 Querying TGraphs – GReQL

Complementing the graph-based approach for the definition of traceability relationships based on grUML, its identification during software development using MOLA transformations, its recording as a TGraph, and its utilization and maintenance using algorithms on TGraphs, a *TGraph query language*, *GReQL* (*Graph Repository Query Language*) [41] is provided. GReQL allows for the retrieval of all kinds of information out of the graph repository.

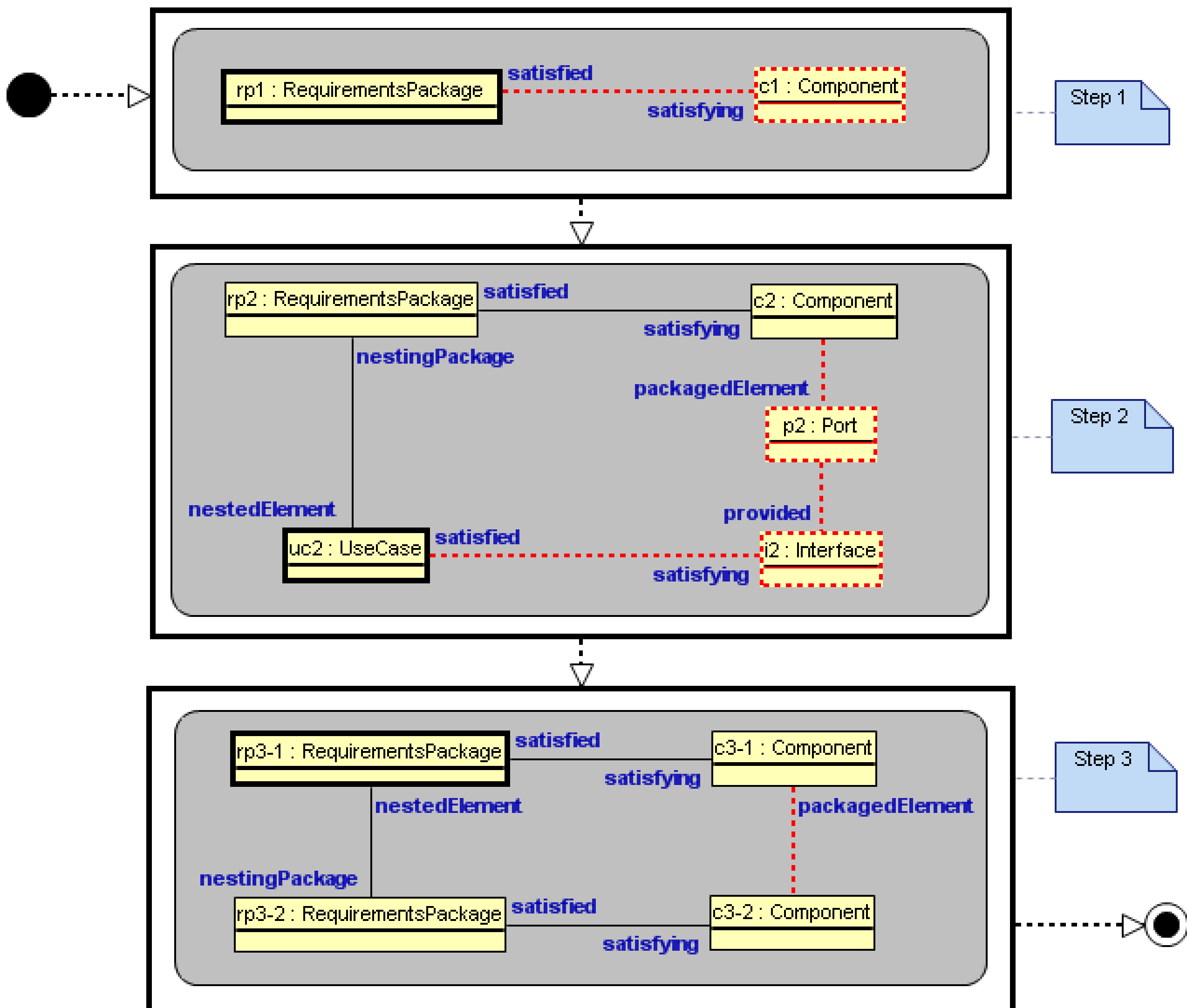

Fig. 5 Transforming requirements to architecture

GReQL is an expression language which describes the data to be retrieved from a given graph by nested expressions. The most important kind of expression is the `from-with-report` expression which returns data described in the `report`-clause as a bag. The bag is constructed by deriving all instances of the variables described in the `from`-clause which fulfill the constraint given in the `with`-clause.

In order to compute the bag of the name attributes of all vertices of type Interface that are connected via a Satisfies link to a vertex of type UseCase, where the latter has the string "`Send Bill`" as its name attribute, a suitable GReQL query might look as follows:

```
from u:V{UseCase}, i:V{Interface}
with u.name = "Send Bill"
     and u <--{Satisfies} i
report i.name
end
```

Expressions in GReQL may employ regular path expressions in order to navigate in the graph. Thus, complicated connections in the graph can be described. As an example, the names of all outer Components, i.e. those which are not contained by other Components, are computed by the following query. Here, a further condition is that the Components directly or indirectly contain a Port which provides an Interface satisfying a given UseCase. Note the usage of the Kleene star to denote transitive closure. Though being more powerful, regular path expressions can be roughly compared to the regular-expression-based language presented in [51].

```
from u:V{UseCase}, c:V{Component}
with u.name = "Receive Payment"
     and u <--{Satisfies}<--{Provides}
           <--{Owns}* c
     and indegree(c) = 0
report c.name
```

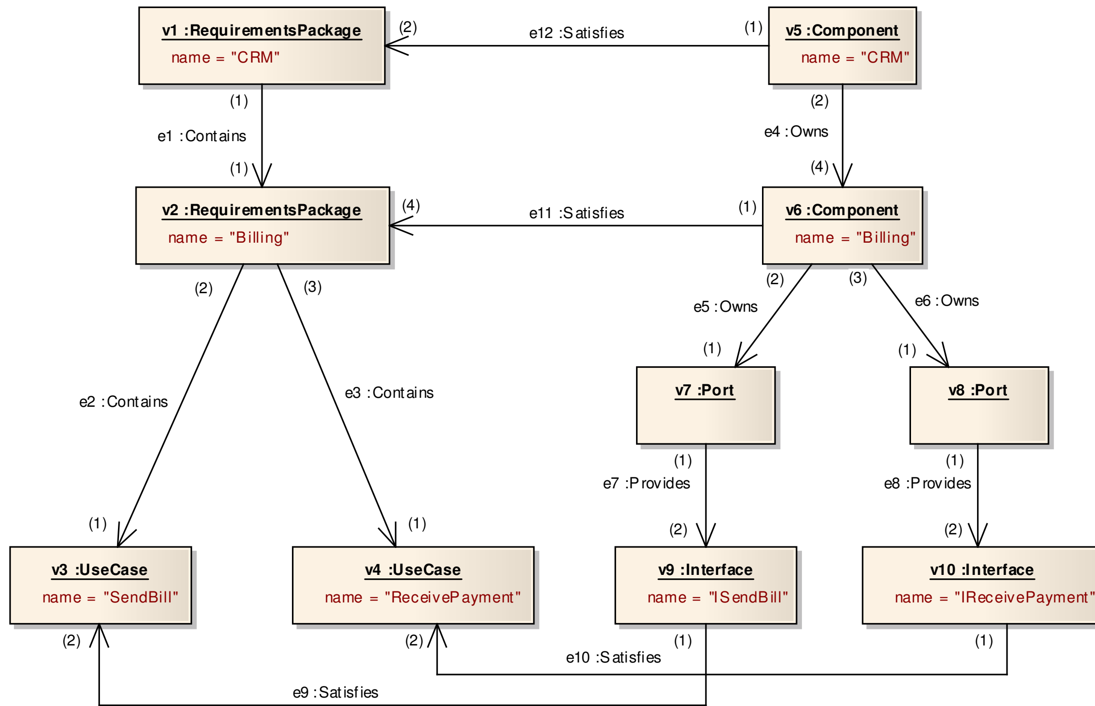

Fig. 6 Resulting graph after transformation

GReQL queries are evaluated by first parsing them and representing their abstract syntax as directed, acyclic TGraphs. Subsequent to potential optimizations, syntax graphs are evaluated by synthesizing the results of specific vertices from the results of their child vertices. In order to evaluate regular path expressions, *nondeterministic finite automata* (*NFA*s) are built based on the path expressions. Then, these NFAs are transformed to *deterministic finite automatons* (*DFA*s). Although theoretically, the number of states of resulting DFAs may explode exponentially, the used algorithms are known to be benevolent in practice [25]. So this will generally only happen with artificial examples. Finally, the DFAs are used to drive the traversal of the graph and to mark relevant vertices and edges [20].

GReQL is an elaborate language coming with an optimized query evaluator that works on TGraphs [30] and is used in software engineering, especially in *reengineering* contexts [41].

In the following sections, the TGraph approach and its accompanying technologies for transformation and querying are applied to support the traceability-related activities identified in section 2.

4 Defining and Recording Traceability Information

As a first building block for a comprehensive and seamless traceability approach, means to define and to record traceability information on any abstraction level must be available. These two activities are closely related because depending on the defined characteristics of traceability relationships, specific requirements are imposed on the recording technology.

In section 4.1, the *Traceability Reference Schema* (*TRS*), a reference schema, or reference metamodel, for traceability is introduced spanning all abstraction levels of a software development process. Section 4.2 describes the repository concept as a means of externally recording traceability information and details the implementation of a concrete, graph-based repository technology. In section 4.3, it is shown how so-called *software cases* conforming to an application-specific adaptation of the TRS, are stored in a graph-based repository in ReDSeeDS.

4.1 The Traceability Reference Schema

Reference models in general are models which describe characteristic properties of categories of systems and serve as reference for specific models representing these systems [61]. In accordance with further goals and characteristics of reference models, especially universality and extensibility, the *TRS* represents a generally applicable reference metamodel for defining traceability relationships and traceable entities. Spanning different levels of abstraction from requirements specification to source code, test cases, and documentation, the TRS is intended to be extended and adapted to specific applications. Since the TRS is modeled using grUML, traceability relationships are stored as TGraph edges which may be specialized and may carry attributes.

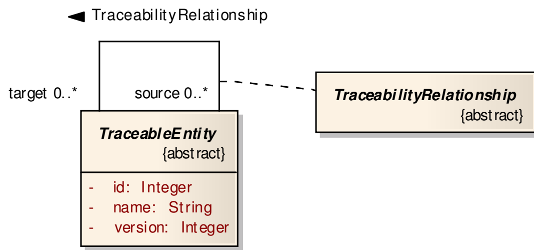

Fig. 7 The core package of the TRS

Adapted from the requirements reference metamodel in [54], the core package of the TRS depicted in figure 7 distinguishes between TraceableEntities on the one hand and TraceabilityRelationships connecting two TraceableEntitys in the roles of source and target on the other hand. While TraceableEntity is defined as a vertex class, an association class is used to define TraceabilityRelationship, leading to the definition of an edge class.

The entities package of the TRS, illustrated in figure 8, contains specializations of TraceableEntity covering different levels of abstraction. Figure 8 shows a possible selection of such specializations. Depending on the specific application, the generalization hierarchy of TraceableEntity is to be modified accordingly. The specializations of Requirement and CodeElement exemplify how to refine the TRS in order to achieve a finer level of granularity.

Concrete traceability relationship types are defined by specializing TraceabilityRelationship in the relationships package, illustrated in figure 9. According to the grUML semantics, since TraceabilityRelationship denotes an edge class, all specializations in figure 9 define edge classes as well. Similar to TraceableEntities, these predefined traceability relationship types are subject to modifications when adapting the TRS to suit specific applications.

Note that using edge classes for modeling traceability relationship types implies a restriction to binary relationship types. Alternatively, if n-ary relationship types are needed, it is possible to model TraceabilityRelationship and its specializations as vertex classes together with edge classes connecting to the participating TraceableEntities. For technical reasons, such an approach is pursued in section 4.2.

Relationship types may be restricted with respect to the entity types they are able to connect. The example in figure 10 shows that traceability relationships of type IsResponsibleFor connect any TraceableEntity to the Stakeholders responsible for their creation or maintenance. Since TGraphs have to conform to their respective schema (cf. section 3.2), instances of IsResponsibleFor must always be incident to a Stakeholder. Besides, the relationship type features two attributes: established and rationale, holding the relationship's creation time and a justification for its existence, respectively.

Since queries can be interpreted as constraints on metamodel instances, further constraints may be introduced with the help of GReQL. For instance, it becomes possible to specify that two given Requirements must not be simultaneously related via a ConflictsWith and a Refines relationship.

The TRS can be roughly compared to the *model weaving metamodel* [17]. However, the intention of the latter is to facilitate the representation of relationships between elements of different models. These relationships not only comprise traceability relationships, but also include relations expressing composition, interoperability, or data integration aspects. Thus, the model weaving metamodel can be considered to be broader in that sense. In contrast to that, the TRS with its entities package is specifically suited for modeling artifacts and fragments of artifacts to be traced.

Similar to the TRS, the TML advocated in [18] also allows for the definition of application-specific traceability metamodels. However, while the TRS represents a metamodel conforming to the grUML meta-metamodel, supposed to be *adapted* to specific needs, the TML takes a different approach: by being a metamodeling language itself, specific traceability metamodels are *instantiated*. Although TML does not support arbitrary additional constraints such as facilitated by GReQL for models adapted from the TRS, the developers of the TML identified common constraints for traceability relationships which can be enabled or disabled using special metaclass attributes. This approach relieves users of dealing directly with the specifics of the underlying constraint language.

4.2 Implementation – Graph-based Repositories

Adaptations of the TRS reference schema are used for defining which traceability relationships are expected to be identified in a given software engineering project.

grUML schemas are regarded as being *tool-ready*, i.e. they serve as the data schema language in tools. This property is hard to achieve with metamodels conforming to full CMOF and was the main reason for the introduction of EMOF. Similarly, grUML was purposely defined as UML subset to provide tool-readiness. Thus, all relevant information on software engineering artifacts can be stored in a *fact repository* directly corresponding to a grUML schema, keeping the abstract syntax of all relevant documents *and* their interconnections by traceability relationships. In contrast, the original artifacts are usually kept in an *artifact repository*, such as SVN.

The repository is populated by a set of *fact extraction tools* which derive the facts from the respective artifacts, effectively creating a graph acting as proxy for the original artifacts. Fact extractors are usually fast enough to be applied regularly to the artifact base in order to generate the corresponding facts, but there are also incremental approaches which replace artifacts' subgraphs in

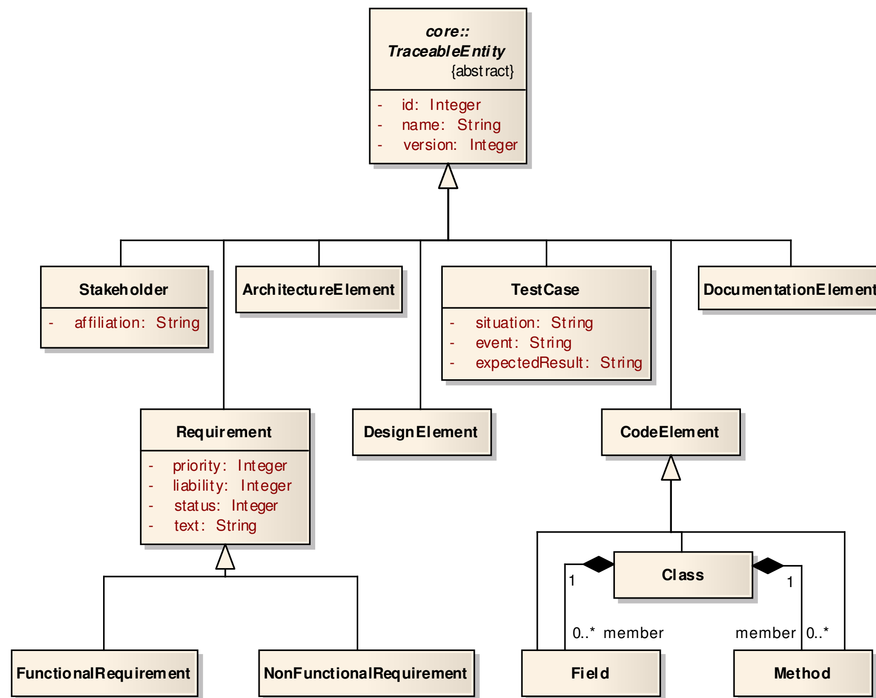

Fig. 8 The entities package of the TRS

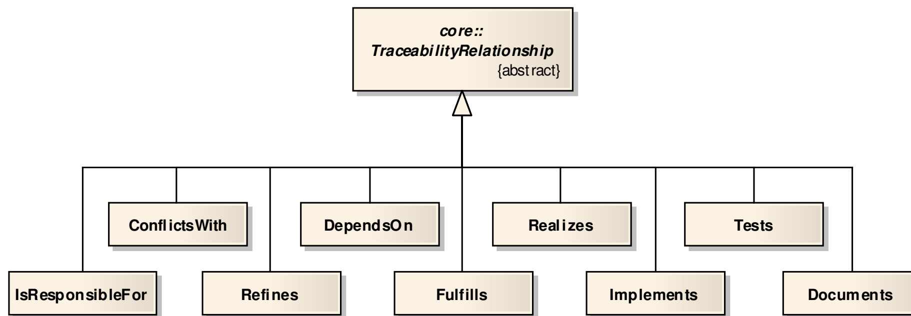

Fig. 9 The relationships package of the TRS

the fact repository [38] on the fly. On the downside, fact extractors are often not trivial to implement. Provided that the repository is to store the structure of artifacts having many different forms of representation, the effort of developing suitable extractors is not to be underestimated.

JGraLab is especially suited to implement fact repositories, based on grUML as its schema language. Since grUML is a proper subset of UML supplied with formal graph semantics, any standard-compliant UML tool can be used to edit schemas. Only minor transformations, e.g. replacing associations between association classes or dependencies between associations, which contradict a pure graph interpretation, by appropriate new classes, are necessary to make MOF-like metamodels tool-ready using grUML.

If a specific algorithm needs to work on the "native", i.e. non-graph, form of representation of an artifact, the original artifact kept in the artifact repository can still be used. However, artifacts' representations in the fact repository can be very fine-granular, only restricted by available hardware resources. So, these algorithms can most probably be adapted to work on the repository instead of directly on the original artifact.

In ReDSeeDS, fact extraction is done by using the API of a UML tool[4] in order to convert UML models into TGraphs. As the result, one integrated graph is delivered which contains all information relevant for retrieving and utilizing traceability (cf. section 6).

Using grUML and JGraLab as repository technology allows for the use of GReQL as repository query language which is especially suited to support transitive closure efficiently using its regular expressions. (see section 6.2 and 6.3)

[4] Prototypically, *Enterprise Architect* by Sparx Systems is used – `http://www.sparxsystems.com`.

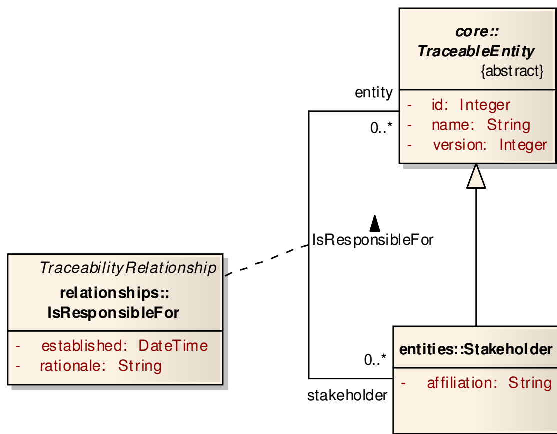

Fig. 10 The IsResponsibleFor traceability relationship type

4.3 Application – Software Cases in ReDSeeDS

In ReDSeeDS, so-called *software cases* [56] comprise all artifacts created in conjunction with single development projects. A small simplified excerpt of the schema of the *Software Case Language* (*SCL*) used to formulate such software cases is shown in figure 11. For modeling entities on the four supported levels of abstraction – requirements specification, architecture, detailed design, and source code – the SCL integrates three sublanguages: the *Requirements Specification Language* (*RSL*) [3], *UML* for creating architecture and detailed design, and *Java*. MOLA is employed for performing model transformations from requirements to architecture to detailed design and finally to source code (cf. section 5).

Comparing the SCL schema to the TRS, the SCL can be regarded as an adaptation of the TRS: SCLElement and TraceabilityRelationship in the SCL can be directly mapped to TraceableEntity and TraceabilityRelationship in the TRS. The metamodels of RSL, UML, and Java are plugged into the schema by modeling the topmost classes of the respective generalization hierarchies, RSLElement, Element from UML, and JavaElement, as specializations of SCLElement.

However, concerning traceability, note that instead of using EdgeClasses as in the TRS, a TraceabilityRelationship is modeled as a VertexClass together with two EdgeClasses connecting it to the SCLElements to be related. This design decision was made due to the technical integration of JGraLab with the transformation language MOLA (cf. section 5.2). Using alternative transformation approaches supporting edges as first-class citizens, a direct adaptation of the TRS would become possible. MOLA was chosen here because it constitutes the general transformation infrastructure in ReDSeeDS. Furthermore, SCL contains some relationships which are not relevant for traceability, thus necessitating the generalization of TraceabilityRelationship by SCLRelationship.

Concrete inter-level traceability relationship types are Satisfies, IsDependentOn, and Implements, connecting an RSLElement with a satisfying architecture UML Element, an architecture UML Element with a depending detailed design UML Element, and a detailed design UML Element with an implementing source code JavaElement, respectively. Examples for intra-level traceability relationship types are the subclasses of RequirementRelationship for connecting RSLElements.

Validation results in ReDSeeDS showed that industry partners positively perceived SCL as a suitable language for modeling requirements, architecture, and detailed design of a software system, including interconnecting traceability relationships. More precisely, the variety of different textual and diagrammatic forms of requirements representation provided by RSL were found to be of benefit, while integration of UML eased modeling of architecture and design to a great extent [40].

Since JGraLab is employed as common repository technology in ReDSeeDS, it is also used for recording traceability information. The choice of JGraLab over other technologies such as relational databases and ontologies is based on various criteria relevant for the project [8]. Among them are the possibility to integrate with model transformation facilities by MOLA, the suitability for calculating similarity between requirements, and the expressiveness of available querying mechanisms.

5 Identifying and Maintaining Traceability Information

By defining a schema for traceability information and selecting an adequate recording infrastructure, the prerequisites for instantiating concrete traceability relationships are met. Subsequently, identification techniques have to be applied in order to create a population of such relationships, be it either manually or (semi-)automatically. As mentioned in section 2.2, similar techniques can be used to maintain already existing relationships. In the following, identification and maintenance with the help of model transformations are investigated. Other approaches, for example relying on information retrieval techniques, could also be applied by a component providing such functionality.

Section 5.1 generally introduces the identification and maintenance of traceability relationships in a model-driven context, i.e. employing model transformations. In section 5.2, the technical integration of MOLA and JGraLab for traceability information is described. Finally, section 5.3 exemplifies the application of the approach in ReDSeeDS by automatically generating traceability relationships between requirements, architecture, and design during software development.

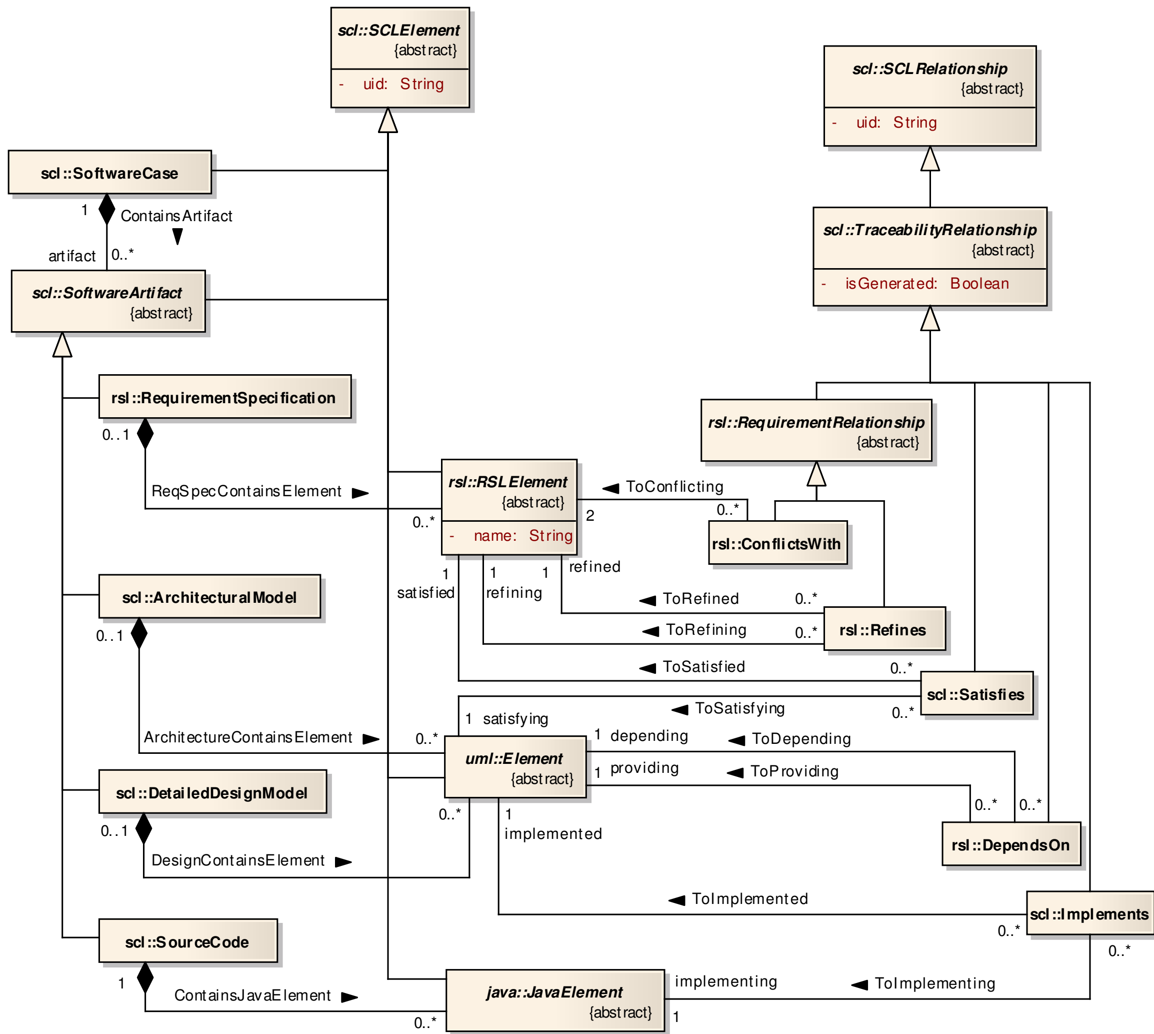

Fig. 11 Excerpt of the SCL schema

5.1 Identifying and Maintaining Traceability Relationships with Model Transformations

There exist various concepts for using model transformations to support the automatic identification of traceability relationships. A closer look at these concepts reveals that they rely on the generation of traceability relationships *during* model transformations. However, another possible approach is to derive traceability relationships *subsequent to* model transformation, based on an analysis of the employed transformation rules.

Clearly, it is desirable to automatically create relationships between source and target entities of model transformations, without users having to manually encode their generation within transformation rules. Regarding this concept, there exists an approach to automatically insert relationship-generating code in ATL transformation rules [33]. With other approaches such as OMG's QVT [49], relationship generation is done implicitly, i.e. it is not even visible within the transformation rules but it is done in the background by the transformation engine. However, many transformation concepts still lack a dedicated support for traceability [1].

Maintenance of traceability relationships when using model transformations is usually triggered by the reexecution of transformation rules upon the change of one of the entities in the source model. However, this results in discarding the previous target model and generating a new one, naturally entailing the regeneration of all traceability relationships. A more sophisticated approach is to selectively update only those entities of the target model which are affected by the change, referred to as *incremental update* by QVT. Existing relationships between changed source and target entities could then be updated accordingly. Incremental pattern matching techniques [6] may help to enhance the performance of this approach. A still open question concerning such updates is how to deal with entities changed manually in the target model. The most straightforward approach, i.e. asking users if the entity shall be regenerated or not, does not scale with larger models.

5.2 Implementation – Using MOLA for Traceability Relationship Generation and Maintenance

The usage of the TGraph approach as a foundation for handling traceability information, especially with the intention to identify traceability relationships based on model transformations, necessitates the incorporation of an adequate transformation language. In ReDSeeDS, the MOLA transformation language and its underlying technology proved to be a suitable candidate for an integration with JGraLab. Besides some technical issues, the main reason for this decision was that the meta-metamodel of MOLA is similar to EMOF [36], whose modeling power can in turn be compared to grUML.

In EMOF, however, there is no direct equivalent for attributed EdgeClasses which may even be part of a generalization hierarchy. Therefore, as was shortly explained in section 4.3, it is necessary to model binary traceability relationship as a VertexClass in conjunction with two EdgeClasses connecting it to the traceable entities to be related (cf. figure 11). This allows for a corresponding representation in EMOF by employing Classes with their respective Properties.

Having a schema usable by both JGraLab and MOLA, transformation procedures in MOLA can be compiled directly to work on JGraLab as repository, thus achieving a tight integration of both technologies and avoiding the need to maintain two separate repositories [57].

For a detailed MOLA transformation example, refer to figure 5.

5.3 Application – Generating Traceability Relationships in ReDSeeDS

The ReDSeeDS project features the automatic transformation of software cases' requirements specifications to architectural models and further on to detailed design models and source code. Generally, this transformation-based approach is applicable for any architectural style. However, the transformation of a requirements specification to the common *4-tier architecture* has been chosen for a prototypical development in ReDSeeDS. This architecture consists of a presentation tier providing the user interface, an application logic tier for guiding control flow, a business logic tier for processing data, and a data access tier for direct interaction with the data source, e.g. a database system.

A simplified example illustrating the creation of an application tier Component together with its Interfaces was already described in section 3. As shown in figure 6, architecture entities and their originating entities on the requirements specification level are linked by Satisfies traceability relationships. Other transformations from requirements to architecture include the generation of business tier components from the requirements specification's vocabulary and the creation of application and business logic tier interface operations from RSL scenario descriptions [12].

Going on to detailed design, transformations add further details to the architecture model, such as factory classes and implementation classes for the previously generated interfaces. Here, DependsOn instances serve to represent traceability relationships between corresponding architecture and design entities. Finally, instances of Implements denote traceability relationships between Java source code fragments and their originating detailed design entities. Altogether, ReDSeeDS features about 40 MOLA procedures realizing these transformations, each one consisting of several rules [37].

Upon the validation of the transformation approach in ReDSeeDS, it turned out that execution of the predefined MOLA procedures generally result in a well-structured and understandable architecture. However, adaptation of these existing MOLA procedures and especially the creation of new ones seem to be a challenging task for average users [40]. The benefit of generated traceability relationships in ReDSeeDS is validated by measuring their ability to support the reuse approach. More information can be found in section 6.3.

Maintenance of traceability relationships is envisioned to be eased by marking elements of target models which have been manually changed afterwards. Upon a subsequent change to the source model, developers would be asked if the manually changed target elements shall be overwritten in the course of the reexecution of the transformation [37]. The traceability relationships connecting elements which are not overwritten are candidates for (manual) maintenance. It is up to the developers to decide whether the relationship between source and target element is still valid. However, for large models this approach may require high manual effort and is subject to further improvement.

6 Retrieving and Utilizing Traceability Information

Once a repository has been populated with traceability information, be it by model transformations as described in section 5 or by other techniques, sought-for traceability information can be retrieved for visualization or other utilizations.

Section 6.1 introduces three patterns which categorize "typical" problems observable when dealing with traceability information retrieval, together with possible fields for utilization. In addition to the general problem statements, the patterns also feature generic queries acting as possible solutions. Therefore, section 6.2 further elaborates on the patterns by giving a selection of such queries formulated in GReQL, including some details on the implementation of GReQL. Concluding, section 6.3 shows the concrete retrieval problem addressed in ReDSeeDS and the utilization of the so-called *slice* pattern in order to solve this problem.

6.1 Common Retrieval Patterns

An analysis of various industrial applications for traceability has been conducted in the *MOST* project[5], leading to a collection of requirements towards the traceability approach to be developed in that project. Based on these requirements, often recurring problems dealing with the retrieval of traceability information were identified. It was possible to abstract these problems into three *patterns* for retrieval. A look at traceability-related literature reveals that many of the traceability problems described therein can be mapped to one of these patterns as well. The patterns are called

- **Existence** Pattern
- **Reachable Entities** Pattern
- **Slice** Pattern

The first pattern, named **Existence**, treats whether there exists a path of traceability relationships between any two traced entities out of given sets of such entities. More formally, the condition is fulfilled if, given two sets of traced entities E_{e1} and E_{e2}, there exist $e_{e1} \in E_{e1}$ and $e_{e2} \in E_{e2}$ with a path of traceability relationships between e_{e1} and e_{e2}. Instead of accepting any path between e_{e1} and e_{e2}, in many cases it is required to restrict eligible paths with respect to the sequence, direction, or type of the involved traceability relationships. These constraints are, for example, expressible by regular path descriptions.

An important application area for **Existence** is quality assurance, e.g. to check whether there exists a realizing architecture component for each requirement. Conversely, investigating if every design element or source code fragment can be traced back to a requirement avoids the implementation of unneeded features. Another usage is to test every design class for the existence of a dedicated test case.

Reachable Entities is concerned with the determination of all traced entities which are reachable from a given set of entities: Given a set of traced entities E_{r1}, the set E_{r2} of traced entities reachable from some $e_{r1} \in E_{r1}$ by following a path of traceability relationships is to be computed. Similar to the **Existence** pattern, mostly it is reasonable to impose various constraints on the structure of the paths which are to be taken into account. It is important to understand that only those entities *at the end* of a path are part of E_{r2}. This aspect makes no difference as long as any path is accepted. But restricting the eligible paths to those of length two, i.e. consisting of two traceability relationships, for instance, would result in intermediate entities not to be included in E_{r2}.

Ranging from change management and maintenance to reverse engineering and project management, the variety of applications for the **Reachable Entities** pattern is manifold. Two concrete examples are the detection of Java classes implementing a specific system component and the determination of stakeholders in order to clarify open questions on particular requirements.

The third pattern, **Slice**, resembles the reachable entities pattern, with the distinction that not only the "endpoints" of a path of traceability relationships, but also all intermediate entities lying on that path and their interconnecting relationships are of interest. More precisely, given a set of traced entities E_{s1}, the set of traced entities E_{s2} corresponding to the set of all entities lying on paths starting at some entity $e_{s1} \in E_{s1}$ form a slice. Furthermore, this slice also contains the relationships which connect each element. Naturally, restrictions of the paths to be considered play an important role here, too. Referring to graph terminology, a slice is effectively a subgraph of the graph formed by the entire set of traced entities and their interconnecting traceability relationships.

The breadth of possible applications of the **Slice** pattern strongly overlaps with that of the **Reachable Entities** pattern. However, there exist many specific use cases which profit from the additional information on intermediate entities provided by a slice. A typical usage is the analysis of a traced entity's origins, i.e. to determine which entities have played a role in the creation of that particular entity. Another application, the reuse of software artifacts as pursued by ReDSeeDS, is presented in more detail in section 6.3.

6.2 Implementation – Retrieval with GReQL

GReQL, the Graph Repository Query Language, is tightly integrated with JGraLab and consequently the main candidate for retrieval of traceability information stored in a TGraph-based repository. As sketched in the following, GReQL proves to be well-suited for formulating queries capable of dealing with the problems represented by the three traceability retrieval patterns. The example queries are based on the transformed graph in section 3.3.

Starting with the **Existence** pattern, the following GReQL query checks if every UseCase can be traced to an Interface by a Satisfies relationship:

```
forall u:V{UseCase} @ exists i:V{Interface}
    @ u <--{Satisfies} i
```

This query does not make use of from-with-report expressions, but directly uses a *quantified expression* returning a boolean value. Obviously, looking at the graph in figure 6, the result of this query is *true*.

When applying the **Reachable Entities** pattern, the GReQL feature *forward vertex set* can be employed: The query below returns all vertices which can be reached from vertex v1 by following a path of arbitrary length whose edges are of any type and direction.

[5] http://www.most-project.eu

```
using v1:
v1 <->*
```

Naturally, this will result in all vertices, i.e. traced entities, somehow reachable from v1 to be returned by that query. Note that v1 does not directly represent the vertex in the example graph, but that it is a *variable* to which that vertex has been previously assigned. With the using clause, such variables can be used in GReQL queries. The binding of variables to values can be done by using the JGraLab API. For reasons of brevity, a source code fragment illustrating this is not presented here.

The expressiveness of regular path descriptions supported by GReQL is useful for narrowing the selection of paths to be taken into account when applying the **Reachable Entities** pattern:

```
using v6:
v6 -->{Owns}-->{Provides}
```

This query serves to get the set of Interfaces {v9, v10} which is provided by Ports owned by the Component v6.

For computing *slices*, GReQL offers a dedicated function called slice, taking a set of vertices and a regular path expression as parameters. In analogy with the program slicing approach [60], the combination of these input parameters is called *slicing criterion*:

```
using v3, v4:
slice(set(v3, v4),
    <--{Satisfies}<--{Provides}<--{Owns})
```

The slice returned by the query above yields the UseCases v3 and v4 together with their satisfying Interfaces, the Ports providing them, and the owning Component. This corresponds to the subgraph consisting of the vertex set {v3, v4, v6, v7, v8, v9, v10} and the edge set {e5, e6, e7, e8, e9, e10}.

6.3 Application – Slicing in ReDSeeDS

The slicing approach can be employed for finding reusable software artifacts in ReDSeeDS. The slicing criterion consists, on the one hand, of a given set reqs, denoting traced entities representing requirements of a past software case identified to be similar to requirements of a current software case. On the other hand, the regular path expression is tailored to retrieve all entities of the past software case which are responsible for realizing one or more of the given requirements.

ReDSeeDS distinguishes between three different notions of a slice: *maximal slice*, *minimal slice*, and *ideal slice* [3]. While maximal slices include every traced entity somehow related to a requirement in reqs, minimal slices almost exclusively consider inter-level traceability relationships, only taking into account intra-level relationships on the requirements level. A GReQL query computing such a minimal slice looks as follows:

```
using reqs:
slice(reqs , (<--{ToRefined}-->{ToRefining})*
    <--{ToSatisfied}-->{ToSatisfying}
    <--{ToDepending}-->{ToProviding}
    <--{ToImplemented}-->{ToImplementing})
```

As can be gathered from this query, the path expression considers inter-level traceability relationships, i.e. Satisfies, DependsOn, and Implements, as well as the Refines intra-level relationship in order to involve requirements which do not belong to those in the reqs set, but are closely related to them (cf. figure 6).

However both, maximal and minimal slices, are likely to not meet users' expectations: Maximal slices probably are equivalent to the whole software case. Minimal slices ignore entities on the architecture, design, or code level which are not directly linked to some requirement, but which are important because entities within the minimal slice depend on them.

Therefore, the concept of an ideal slice tries to formulate a suitable path expression for capturing entities which are needed for the proper functioning of entities linked to requirements by inter-level relationships. Essentially, an ideal slice considers nesting relationships and dependency relationships such as package imports or call relationships. Thus, it is ensured that components of traceable entities as well as other entities providing required functionality are included in the slice. Due to the complexity of the SCL schema, of which figure 6 shows only an excerpt and omits many intra-level relationships, these path expressions are very intricate. More information can be found in [10].

On the one hand retrieved ideal slices are expected to be precise enough to include traced entities which are needed to achieve a proper functionality of entities directly related to the requirements. But on the other hand they ignore all other non-related entities. ReDSeeDS validation results regarding this are to be published in [7].

7 Conclusion

This paper introduces the TGraph-based approach to formalize and implement traceability information in software engineering projects. The TGraph approach for traceability management was developed and realized in the ReDSeeDS project which aims at the support of software reuse based on similarity of requirement definitions. In order to derive reusable software elements, a comprehensive and seamless approach for all traceability-related activities was requested. TGraph-based technology provides a comprehensive and smoothly integrated means to traceability management comprising all traceability-related activities during software development and maintenance.

TGraph-based metamodeling, using an adaptable and extensible reference structure for defining traceability information via grUML-class diagrams provides a formal description of project relevant traceability data. Simultaneously, these metamodels *define* graph-based data structures to *record* traceability information in the JGraLab graph repository. Graph-based transformations, using the MOLA transformation engine, are used to *identify* and *maintain* traceability information, and graph queries, using the GReQL graph query engine, support efficient and comprehensive *retrieval* and *utilization* of traceability interrelations.

Applying the approach to realistic industrial development projects, contributed by various industrial partners in the ReDSeeDS project, facilitated the development and validation of an applicable technique for traceability management. Results of the empirical validation in ReDSeeDS showed that the concepts for definition and recording are well-accepted by the industrial partners. The approach for identifying and maintaining traceability relationships is also promising with respect to predefined transformation rules, whereas the adaptation and creation of rules in order to modify the relationship generation seem to require much training. Validation of the retrieval and utilization mechanism is still in progress.

Although ReDSeeDS is clearly set in a model-driven context, the approach is supposed to be applicable to other methodologies, even if traceability relationships are not identified via model transformations. Concepts such as the definition of traceability information on the basis of the TRS or retrieval with GReQL can be used with “traditional” development processes. Of course, identification then has to be conducted with the help of other techniques.

Existing limitations of the presented approach can be assigned to three aspects. Firstly, there is the purely technical inability of MOLA to deal with graph edges as first-class citizens that leads to the necessity of modeling binary traceability relationships as vertex classes. Secondly, the approach only considers identification and maintenance of relationships by model transformations yet, while neglecting other techniques, e.g. based on information retrieval. Thirdly, the current maintenance approach is very simple, relying on complete reexecution of transformations and demanding manual work from the developers, consequently hampering scalability.

Addressing the individual limitations, future work is to possibly replace MOLA by a language specifically developed for transforming TGraphs [21], thus eliminating the need for the reification of binary relationships. Furthermore, it is feasible to integrate tools for identifying and maintaining traceability relationships with the graph-based fact repository, either by implementing new fact extractors or by relying on an exchange format such the *Graph eXchange Language* (*GXL*) [29]. This would allow to also rely on other techniques than transformations only. Finally, the possibility of integrating incremental approaches for graph transformations [6] has to be examined in order to increase efficiency when changing source models. In addition, the suitability of integrity constraints for relieving developers from manual effort in identifying and maintaining relationships deserves further investigation. Comparable to the approach taken in [18], the detection of missing relationships or of existing relationships to be updated due to some change to traced artifacts could be assisted by using GReQL as a constraint language, i.e. employing queries which need to evaluate to true in order for a model to be valid. Provided that proper constraints are formulated for each relationship type, violation of such a constraint indicates that a traceability relationship is missing or no longer valid.

Summing up, this paper promotes a comprehensive and seamless approach for supporting traceability activities. Based on graph technology used for defining traceability information, the approach features a repository approach for recording this information. A repository represents a common pool of data external tools may be integrated with. Such an integration with the MOLA transformation tool allows for transformations supporting identification and maintenance of traceability relationships. The graph query language GReQL is used for retrieval and utilization of traceability information. The overall approach is validated in the ReDSeeDS project.

Acknowledgements This work has been partially funded by the European Commission within the 6th Framework Programme project ReDSeeDS, no. IST-2006-33596, `http://www.redseeds.eu`, and the 7th Framework Programme project MOST no. ICT-2008-216691, `http://most-project.eu`.

References

1. N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.
2. Netta Aizenbud-Reshef, Richard F. Paige, Julia Rubin, Yael Shaham-Gafni, and Dimitrios S. Kolovos. Operational Semantics for Traceability. In *ECMDA Traceability Workshop (ECMDA-TW) 2005 Proceedings*, 2005.
3. Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, and Tomasz Straszak. Can Precise Requirements Models Drive Software Case Reuse? In *Proceedings of the 2nd International Workshop on Model Reuse Strategies (MoRSe 2008)*, pages 27–34, 2008.
4. Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.
5. Hazeline U. Asuncion, Frédéric François, and Richard N. Taylor. An End-To-End Industrial Software Traceability Tool. In *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC-FSE '07)*, pages 115–124, 2007.

6. Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental Pattern Matching in the VIATRA Model Transformation System. In *GRaMoT '08: Proceedings of the Third International Workshop on Graph and Model Transformations*, pages 25–32, 2008.
7. Daniel Bildhauer, Jürgen Ebert, Tassilo Horn, Lothar Hotz, Stephanie Knab, Volker Riediger, and Katharina Wolter. Final Software Case Query Language Definition. Project Deliverable D4.1.2, ReDSeeDS Project, 2009. To appear in September 2009.
8. Daniel Bildhauer, Jürgen Ebert, Volker Riediger, Thorsten Krebs, Markus Nick, Hannes Schwarz, Audris Kalnins, Elina Kalnina, Markus Nick, Sören Schneickert, Edgars Celms, Katharina Wolter, Albert Ambroziewicz, and Jacek Bojarski. Repository Selection Report. Project Deliverable D4.4, ReDSeeDS Project, 2007.
9. Daniel Bildhauer, Jürgen Ebert, Volker Riediger, and Hannes Schwarz. Using the TGraph Approach for Model Fact Repositories. In *Proceedings of the 2nd International Workshop MoRSe 2008: Model Reuse Strategies – Can requirements drive reuse of software models?*, pages 9–18, 2008.
10. Daniel Bildhauer, Jürgen Ebert, Volker Riediger, Katharina Wolter, Markus Nick, Andreas Jedlitschka, Sebastian Weber, Hannes Schwarz, Albert Ambroziewicz, Jacek Bojarski, Tomasz Straszak, Sevan Kavaldjian, Roman Popp, and Alexander Szep. Software Case Marking Language Definition. Project Deliverable D4.3, ReDSeeDS Project, 2007.
11. Boris Böhlen. *Ein parametrisierbares Graph-Datenbanksystem für Entwicklungswerkzeuge*. Shaker Verlag, 2006.
12. Jacek Bojarski, Tomasz Straszak, Albert Ambroziewicz, and Wiktor Nowakowski. Transition from precisely defined requirements into draft architecture as an MDA realisation. In *Proceedings of the 2nd International Workshop MoRSe 2008: Model Reuse Strategies – Can requirements drive reuse of software models?*, pages 35–42, 2008.
13. Jane Cleland-Huang, Carl K. Chang, and Mark Christensen. Event-Based Traceability for Managing Evolutionary Change. *IEEE Transactions on Software Engineering*, 29(9):796–810, 2003.
14. Gilberto Cysneiros and Andrea Zisman. Traceability and Completeness Checking for Agent-Oriented Systems. In *Proceedings of the 2008 ACM symposium on Applied computing (SAC '08)*, pages 71–77, 2008.
15. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
16. Jeremy Dick. Rich Traceability. In *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering*, 2002.
17. Marcos Didonet Del Fabro, Jean Bézivin, and Patrick Valduriez. Weaving Models with the Eclipse AMW plugin. Eclipse Modeling Symposium, Eclipse Summit Europe 2006, 2006.
18. Nikolaos Drivalos, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes. Engineering a DSL for Software Traceability. In *Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, pages 151–167, 2008.
19. J. Ebert and A. Franzke. A Declarative Approach to Graph Based Modeling. In *Graphtheoretic Concepts in Computer Science*, pages 38–50, 1995.
20. Jürgen Ebert and Daniel Bildhauer. Querying Software Abstraction Graphs. In *Proceedings of Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, 2008.
21. Jürgen. Ebert and Tassilo Horn. The GReTL Transformation Language. Internal Report. To appear, 2009.
22. Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO. Generic Understanding of Programs - An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002. http://www.elsevier.nl/locate/entcs/volume72.html.
23. Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph Technology in Reverse Engineering, The TGraph Approach. In Rainer Gimnich, Uwe Kaiser, Jochen Quante, and Andreas Winter, editors, *Proceedings of the 10th Workshop Software Reengineering (WSR 2008)*, volume 126 of *GI Lecture Notes in Informatics*, pages 67–81, 2008.
24. Angelina Espinoza, Pedro P. Alarcón, and Juan Garbajosa. Analyzing and Systematizing Current Traceability Schemas. In *Proceedings of the 30th Annual IEEE/-NASA Software Engineering Workshop SEW-30 (SEW '06)*, pages 21–32, 2006.
25. Rob Glabbeek and Bas Ploeger. Five Determinisation Algorithms. In *Proceedings of the 13th International Conference on Implementation and Applications of Automata (CIAA '08)*, pages 161–170, 2008.
26. Arda Goknil, Ivan Kurtev, and Klaas van den Berg. Change Impact Analysis based on Formalization of Trace Relations for Requirements. In *ECMDA Traceability Workshop (ECMDA-TW) 2008 Proceedings*, pages 59–75, 2008.
27. Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An Analysis of the Requirements Traceability Problem. In *Proceedings of the 1st International Conference on Requirements Engineering*, pages 94–102, 1994.
28. Mark Grechanik, Kathryn S. McKinley, and Dewayne E. Perry. Recovering And Using Use-Case-Diagram-To-Source-Code Traceability Links. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering (ESEC-FSE '07)*, 2007.
29. Richard C. Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. GXL: A Graph-Based Standard Exchange Format for Reengineering. *Science of Computer Programming*, 60(2):149–170, 2005.
30. Tassilo Horn. *Ein Optimierer für GReQL2*. Grin Verlag, 2009.
31. Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.
32. IEEE. *IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990*, 1990.

33. Frédéric Jouault. Loosely Coupled Traceability for ATL. In *ECMDA Traceability Workshop (ECMDA-TW) 2005 Proceedings*, pages 29–37, 2005.
34. Hermann Kaindl. The Missing Link in Requirements Engineering. *SIGSOFT Software Engineering Notes*, 18(2):30–39, 1993.
35. Audris Kalnins, Janis Barzdins, and Edgars Celms. Model Transformation Language MOLA. In *Model Driven Architecture: Foundations and Applications (MDAFA)*, 2004.
36. Audris Kalnins, Edgars Celms, and Agris Sostaks. Tool Support for MOLA. In *Proceedings of the Workshop on Graph and Model Transformation (GraMoT)*, pages 162–163, 2005.
37. Audris Kalnins, Elina Kalnina, Edgars Celms, Agris Sostaks, Hannes Schwarz, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Sevan Kavaldjian, and Jürgen Falb. Reusable Case Transformation Rule Specification. Project Deliverable D3.3, ReDSeeDS Project, 2007.
38. Manfred Kamp. Managing a Multi-File, Multi-Language Software Repository for Program Comprehension Tools — A Generic Approach. Technical Report 1/98, Universität Koblenz-Landau, Institut für Informatik, 1998.
39. Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. GRAS, a graph oriented (software) engineering database system. *Information Systems*, 20(1):21–51, 1995.
40. Thorsten Krebs, Wiktor Nowakowski, Audris Kalnins, and Elina Kalnina. Modelling and Transformation Language Validation Report. Project Deliverable D3.4, ReDSeeDS Project, 2007.
41. Bernt Kullbach and Andreas Winter. Querying as an Enabling Technology in Software Reengineering. In C. Verhoef and P. Nesi, editors, *Proceedings of the 3rd Euromicro Conference on Software Maintenance & Reengineering*, pages 42–50, 1999.
42. Ivan Kurtev, Matthijs Dee, Arda Goknil, and Klaas van der Berg. Traceability-based Change Management in Operational Mappings. In *ECMDA Traceability Workshop (ECMDA-TW) 2007 Proceedings*, pages 57–67, 2007.
43. Patrick Mäder, Orlena Gotel, and Ilka Philippow. Rule-Based Maintenance of Post-Requirements Traceability Relations. In *Proceedings of the 16th IEEE International Requirements Engineering Conference*, pages 23–32, 2008.
44. Jonathan I. Maletic, Michael L. Collard, and Bonita Simoes. An XML Based Approach to Support the Evolution of Model-to-Model Traceability Links. In *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 67–72, 2005.
45. Andrian Marcus, Xinrong Xie, and Denys Poshyvanyk. When and How to Visualize Traceability Links? In *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 56–61, 2005.
46. Kurt Mehlhorn, Stefan Näher, and Christian Uhrig. The LEDA Platform of Combinatorial and Geometric Computing. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP '97)*, pages 7–16, 1997.
47. Mikyeong Moon, Heung Seok Chae, Taewoo Nam, and Keunhyuk Yeom. A Metamodeling Approach to Tracing Variability between Requirements and Architecture in Software Product Lines. In *Proceedings of the 7th IEEE International Conference on Computer and Information Technology*, pages 927–933, 2007.
48. Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*, 2007.
49. Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0*, 2008.
50. Francisco A.C. Pinheiro. Formal and Informal Aspects of Requirements Tracing. In *Anais do WER00 - Workshop em Engenharia de Requisitos 2000*, 2000.
51. Francisco A.C. Pinheiro and Joseph A. Goguen. An Object-Oriented Tool for Tracing Requirements. *IEEE Software*, 13(2):52–64, 1996.
52. Klaus Pohl. *Process-Centered Requirements Engineering*. Research Studies Press Ltd., 1996.
53. Balasubramaniam Ramesh and Matthias Jarke. Toward Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001.
54. Hannes Schwarz, Jürgen Ebert, Volker Riediger, and Andreas Winter. Towards Querying of Traceability Information in the Context of Software Evolution. In *10th Workshop Software Reengineering (WSR 2008)*, 2008.
55. Susanne A. Sherba, Kenneth M. Anderson, and Maha Faisal. A Framework for Mapping Traceability Relationships. In *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering*, 2003.
56. Michał Śmiałek. Towards a requirements driven software development system. Poster presentation at MoDELS, 2006.
57. Agris Sostaks and Audris Kalnins. The Implementation of MOLA to L3 Compiler. In *Computer Science and Information Technologies*, volume 733 of *Scientific Papers University of Latvia*. Latvijas Universitate, 2008.
58. George Spanoudakis and Andrea Zisman. Software Traceability: A Roadmap. In S. K. Chang, editor, *Handbook of Software Engineering & Knowledge Engineering: Recent Advances*, volume 3, pages 395–428. World Scientific Publishing Company, 2005.
59. Antje von Knethen and Barbara Paech. A Survey on Tracing Approaches in Theory and Practice. Technical Report 095.01/E, Fraunhofer IESE, 2002.
60. Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
61. Andreas Winter. *Referenz-Metaschema für visuelle Modellierungssprachen*. DUV Informatik. Deutscher Universitätsverlag, 2000.
62. René Witte, Yonggang Zhang, and Juergen Rilling. Empowering Software Maintainers with Semantic Web Technologies. In *Proceedings of the 4th European Semantic Web Conference (ESCW 2007)*, pages 37–52, 2007.