A Pragmatic Approach to Traceability in Model-Driven Development

Markus Aleksy, Tobias Hildenbrand, Claudia Obergfell, Martin Schader, Michael Schwind
University of Mannheim, Schloss, D-68131 Mannheim, Germany
{aleksy, hildenbrand, schader, schwind}@uni-mannheim.de, claudia.obergfell@arcor.de

Abstract:
A common problem in model-driven software development (MDSD) processes is the tracing of requirements across different phases of the software development life cycle and multiple levels of abstraction down to the code level. Because debugging at the model level is not feasible yet, unwanted or unexpected behavior of the executable system needs to be analyzed at the code level at run-time and in a feedback loop must be traced back to and handled at the model level. Thus, traceability is a very important success factor and quality criterion in software engineering and maintenance and especially when developing high-quality model-driven infrastructures. In this paper, we present the conceptual design and prototypical implementation of a lightweight traceability approach which supports tracing requirements across different models and levels of abstraction. While providing support for representing different types of traceability links between design models and implementation details, our approach can easily be integrated into existing MDSD projects without increasing their complexity.

1 Introduction

Model-driven software development aims at raising the level of abstraction of development processes by describing software systems using formal models on different levels of abstraction, which are ultimately used as a basis for automatic code generation (cf. [31], [12]). Ideally, all changes to an existing system are applied to the model level and propagated to the code level by performing transformations and code generation procedures. Software development efforts benefit from this approach in several ways: according to Mellor and Balcer [34], Bézivin [10] as well as Booch et al. [11], the two main goals of MDSD are to improve the robustness of software artifacts to changes applied to a software system [9] and, more important in this context, to increase the level of abstraction, allowing to better deal with the problem of complexity. It is argued that any approach addressing these problems will ultimately result in a reduction of cost and time to market, which are the main selling arguments given by MDSD advocates. However, the optimistic outlook of the MDSD proponents [35] is not shared by all experts in the software engineering discipline (e.g., [25]).

Despite automatic model transformations, establishing traceability is not a trivial in MDSD projects. Due to the large number of artifacts and their interdependecies, this task is time

consuming, tedious and error-prone [20]. A few basic preconditions for effective and efficient traceability management have been identified in theory and practice, most importantly:

- *automatic recovery, validation, and update* of traceability links [5] and
- *tool integration* (in particular with respect to widely used and possibly heterogeneous development environments) [16].

Therefore, we propose an approach that enables and supports traceability in a model-driven context by allowing the creation and management of three types of explicit traceability links: (1) between requirements and model elements, (2) between model elements at different levels of abstraction, and (3) between model elements and code sections. Based on these three types of *explicit* traceability links, *implicit* links between requirements and code sections realizing them can be derived—and thus horizontal and bi-directional traceability, as required by many industry standards (e.g., [14]), can be achieved even in complex distributed development projects. For instance, this traceability information can be exploited to verify that all specified requirements are reified in code.

The approach we suggest has been designed to satisfy both of the previously mentioned requirements, i.e., automation and tool integration. Due to the fact that many publications neglect the importance of tool support for end-to-end traceability (cf. section 6), we have chosen a pragmatic approach, that can be integrated with existing MDSD tool chains and is suitable to interoperate with widely accepted collaboration platforms. Following a fundamental design science research methodology [26], we have implemented a prototypical tool support and integration with existing development environments. The software prototype is used to demonstrate that our approach is applicable in distributed settings, can easily be integrated (in our case with the Eclipse and CodeBeamer development platforms [17, 29]), and can be applied in practice. The prototype also shows that our approach can be automated and does not necessarily require extensive and time-consuming manual trace capturing. To evaluate the novelty and utility of our approach we have conducted an informed comparison with related research (see [26, pp. 85]).

The remainder of this paper is structured as follows: after a fundamental discussion on the role of traceability, in particular in the context of MDSD (section 2), the conceptual design of our prototype is presented in section 3. The main part of this paper (cf. section 4) contains a description of the prototype that has been implemented to demonstrate the applicability of the traceability concept described. This is followed by a discussion of related research efforts as compared to our approach. The paper concludes with a summary of our findings and an outlook on future developments.

2 The Role of Traceability in Model-driven Development

MDSD approaches rely on two basic assumptions: first, that all requirements to a system are fully and precisely reflected by the models, and second, that each model element

is ultimately transformed accurately into executable application code. The first requirement is relaxed by some interpretations of MDSD, such as architecture-centric MDSD approaches [39], that rely on manual completion of the generated code and concepts, i.e., by defining protected code regions. Common to all MDSD approaches is the raised level of abstraction and the problem of tracing requirements and other artifacts from the model level to the source code level, sometimes across several intermediate model representations [1]. To support traceability in MDSD, it is necessary that the relation between each requirement, its representation in the models (e.g., as model elements, such as classes or associations) and the resulting code sections can be captured, managed, and analyzed [16]. Thus, traceability is generally a critical success factor and quality criterion in model-driven development projects.

Model transformation is one of the key features of MDSD: abstract models are transformed into more and more concrete models and ultimately into source code. In the terminology of the Model-Driven Architecture (MDA), a Platform-Independent Model (PIM) is successively transformed into a Platform-Specific Model (PSM), which is then transformed into code. While the idea of a stepwise reduction of the level of abstraction through repeated model transformations is an integral part of the OMG's MDA initiative, other approaches pursue a model-to-code transformation-based strategy, where either no intermediate models are used or where they are not intended to be manipulated by the modeler.

Either way, traceability plays a central role throughout the MDSD process, because it enables developers to maintain an insight on the relationships between various artifacts, on different levels of abstraction, and, in the case of multiple successive transformations, even between individual transformations. For example, traceability between elements of one model can help to identify both dependencies and commonalities (e.g., two model elements based on a given requirement, cf. [1]). In locally distributed MDSD projects, where people at various sites collaborate on different parts of the models, establishing and maintaining traceability becomes even more difficult [37]. When traceability relations are managed correctly and efficiently, however, traceability can substantially support coordination in both distributed and collocated software projects [36].

As already mentioned before, traceability between requirements and their representation in the models is crucial to ensure that the relevant set of requirements is accurately elicited and eventually implemented in the code. Additionally, traceability information constitutes a suitable basis for further transformations and code generation [33]. Moreover, traceability between a model element and the code generated from this element is important for debugging generated code and for program comprehension in general. While syntactic correctness of both input and output artifacts, i.e., models, source code files, or XML configuration data, as well as adherence to a particular metamodel can be verified, it is difficult to trace unexpected behavior of the resulting system back to one of the more abstract model levels.

Finally, traceability between individual transformations is a necessary pre-condition in order to decompose complex model transformations into modular steps. An example can be found in [40], where information about preceding transformations is used in order to determine which transformation steps should be performed later on in the process. MDSD development processes are well-suited to be (locally) distributed, either by partitioning

application development into sub-tasks, or by assigning application and infrastructure development tasks to different teams (cf. [39]).

3 The TRACES Approach

In this section we provide an overview on the concepts our approach is based on and the different kinds of traceability links that are supported.

3.1 Assumptions

We assume that requirements are represented by a textual description (free text) and a unique identifier. This notion also includes requirements captured in collaboration tools, such as CodeBeamer [29]. We also assume that each model element, such as a class or an association, has a unique identifier. This way, requirements and model elements can be referenced unambiguously. No assumptions about the format of these identifiers are made, however, as long as their uniqueness is guaranteed.

Since the field of model-to-model transformations is still an emerging one, our initial prototype is based on a development process where code is directly generated from the models. Nevertheless, these models are both platform-independent and technology-independent. As a basis for our prototype we have used the OMEGA modeling and code generation infrastructure [22, 23], which will be described briefly in section 4 in conjunction with CodeBeamer.

With regard to code generation, it is assumed that the source code is generated in its entirety, i.e., no manual completion of the generated source code is required. As a consequence, each individual code section is based on one ore more specific model elements. Additionally, all relevant model elements, i.e., classes, associations, etc., including their identifiers need to be available at generation time.

3.2 Explicit Traceability Links

Based on the assumptions described in the previous section, traceability between requirements and model elements is achieved by creating detailed design models where each model element references all requirements it represents. Since all requirements have unique identifiers (cf. section 3.1), these references are unambiguous.

Traceability between model elements and code sections is achieved by creating appropriate references during code generation.These references can be introduced into the code and can contain the identifier of the model element a section is based on. That way, unambiguous references to model elements are created in the code. These can be exploited to trace run-time problems back to specific elements in the original model, which means

that changes necessary to solve these problems can be applied at the model level and, thus, debugging in MDSD environments is facilitated.

The creation of explicit traceability links is the responsibility of the modeler, and to some degree of the MDSD infrastructure provider. While the former is responsible for introducing references from requirements to model elements manually when modeling a system, the latter must make sure that the resources used in the MDSD process, such as source code templates and code generators, are prepared to propagate these references across abstraction levels and to introduce them into the generated output. Explicit traceability links represent the developers knowledge about requirements and their realization in the developed system. They consist of references between model elements and requirements descriptions that cannot be introduced automatically. Additionally the introduction of traceability links requires an object model that contains the information on how a particular traceability link is represented throughout the model transformation and code generation process. While the former must be created for each individual model, the latter is part of a reusable infrastructure and only needs to be created once.

3.3 Implicit Traceability Links

In addition to the retrieval of explicit traceability information as described in the previous section, implicit traceability links between requirements and code sections can be automatically derived using explicit links (cf. requirements in section 1). The creation of implicit traceability links can help to gain a more complete insight into a system by showing relationships between artifacts that the developer may not have been aware of.

For example, from the knowledge that model element A is involved in realizing requirements R_1 and R_2 and that code section C is based on model element A, we can conclude that code section C is involved in realizing requirements R_1 and R_2, as well. Thus, we can use the available information, for instance, to check if all requirements are implemented in the application code. Similarly, the knowledge that any two model elements reference the same requirement could be used to derive a traceability relationship between these two model elements.

The creation of implicit or inferred (cf. [1]) traceability links requires an automatic analysis of existing explicit links in order to identify relationships that the developer has not explicitly modeled, but that are of importance, e.g., when performing change impact analyses to existing systems. Our approach supports this kind of analysis because each requirement and each model element is given a unique identifier (cf. section 3.1). Based on these identifiers, implicit links between requirements and code sections and between model elements can be retrieved by parsing models and code.

4 OMEGA TRACES Prototype

We have implemented a prototype to demonstrate the utility and the practical applicability of our approach. The prototype is fully integrated into the OMEGA modeling and code generation infrastructure. Due to the fact that our approach does not rely on being used in conjunction with OMEGA, but can be used with other MDSD tool chains, only a short introduction is provided. For a more comprehensive discussion, the reader is referred to [22] and [24].

4.1 The OMEGA Approach and Architecture

OMEGA (Ontological Metamodel Extension For Generative Architectures) is an approach to model-driven development that is targeted at facilitating the rapid development of domain-specific modeling and code generation tools. The approach draws from Executable UML (cf. [34]), i.e., it uses class and state chart models to describe software systems at an abstract level. It strongly promotes the reuse of code generation artifacts, such as model transformation scripts and source code templates. Instead of using a general-purpose modeling language, OMEGA relies on domain-specific languages, represented using hierarchies of domain metamodels, e.g., for the domain of web applications. The use of metamodel hierarchies has been suggested by Atkinson and Kühne [9]. OMEGA draws from this idea of describing problem domains on various levels of abstraction.

Based on the theoretical concepts outlined here, a prototype has been implemented as an extension to the Eclipse development environment. Even though OMEGA relies on some assumptions that clearly distinguish it from other model-driven software development approaches, our traceability framework does not depend on these assumptions or any other specifics of OMEGA. Therefore, its use is not restricted to the OMEGA environment. Rather, it can be adapted to be utilized in other code generation environments as well (cf. [22]).

The current OMEGA prototype consists of three Eclipse plugins; the core is a generic modeling tool which is supplemented by a metamodel implementation and a simple, template-based code generator. Figure 1 shows an overview of the components that constitute the OMEGA infrastructure; a detailed description can be found in [22].

Based on (1) the OMEGA modeling environment (section 4.2), the following subsections relate the requirements outlined in the previous section to existing tool support, also for ensuing process steps of (2) code generation (section 4.3), (3) explicit and implicit trace capturing and validation (section 4.4) as well as (4) traceability information management and visualization (section 4.5).

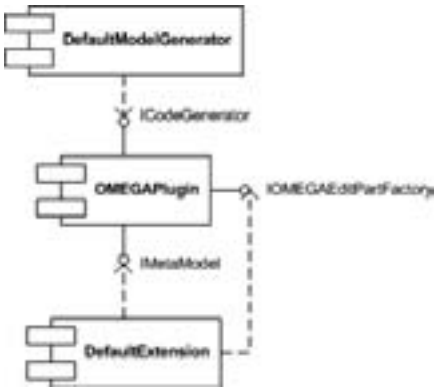

Figure 1: Basic structure of the prototype as described in ([22], p. 142)

4.2 Modeling

The modeling tool consists of a model editor and a number of views which allow a user to edit models viewing specific aspects of them. It supports both static (class diagrams) and dynamic (state chart) models. Additionally, easy to use requirements management, allowing a user to add, edit, or delete project-specific requirements using a graphical user interface (GUI) is supported.

To help automating traceability management, the software handles identifiers for requirements and model elements automatically: whenever a new requirement or model element is created, a unique identifier for that artifact is automatically created. All identifiers are displayed in the GUI. The identifiers of requirements are used when registering or unregistering references to an arbitrary selection of requirements to a model element—which can also be accomplished using the GUI. These references are then stored with the model element in the serialized model.

Models are written to files in a special XML format using the XStream [13] library or, alternatively to XMI, using the Eclipse ECore API ([18]). This way, all model data including model elements as well as references to requirements and information on potential submodels are stored in a single XML file, allowing external tools to retrieve traceability information by parsing the model files.

4.3 Code Generation

The code generator takes a model from the modeling tool as input and transforms it to a format more suitable for code generation than the original one. Based on this "generator model", application code is generated automatically, using a template-based approach [15]. During this transformation, all references to requirements are preserved.

For the purpose of generating executable systems from the generator model, the Velocity Template Engine [8] is used. The template engine has access to all model data using

a context object, which also includes the identifiers of the model elements. Thus, the inclusion of appropriate statements in the templates allows the creation of comments in the source code referencing the model elements each code section is based on.

4.4 Trace Capturing

When using the TRACES tool for capturing traces among requirements and models, the two types of traceability links described in section 3 need to be distinguished. In the following sections, we describe the ways in which *explicit* and *implicit* traceability links are handled in our prototype.

4.4.1 Explicit Traceability Links

Traceability links between requirements and model elements are created using the modeling tool. This is done by adding references to all relevant requirements to the model elements. Figure 2 shows a screenshot of the user interface for associating textually represented requirements with model elements in the development environment. The attribute `DateOfBirth`, selected in the lower section of the screen is needed to realize the requirement of storing the date of birth of every customer.

The relationship between model elements and requirements usually is a many-to-many relationship, meaning that several model elements can reference the same requirement, and that one requirement can be realized by one or more model elements. OMEGA supports this kind of relationship by allowing references to an arbitrary number of requirements for each model element. As figure 2 shows, the list of requirements uses check boxes allowing a modeler to select and add references to an arbitrary number of requirements.

Traceability links between model elements and code sections are created automatically during the code generation process. The associations between model elements and output artifacts are preserved in all internally used intermediate model representations and thus can be introduced into the generated output resources, such as Java source code files. No additional user input is necessary to do this, as the generator model contains all the information needed and the format of the links is defined by the generator templates. While this procedure increases ease of use, it also has a disadvantage: the templates defined for the output artifacts must contain variables that the code generator can replace with references to the model level. Therefore, an initial adjustment of the templates to being used in conjunction with the traceability module is required.

4.4.2 Implicit Traceability Links

OMEGA TRACES supports the extraction of implicit traceability links as described in section 3.3 by serializing the models in a special XML format, which also contains the explicit links between model elements and requirements.

Currently, the prototype relies on external tools for the extraction of implicit traceability

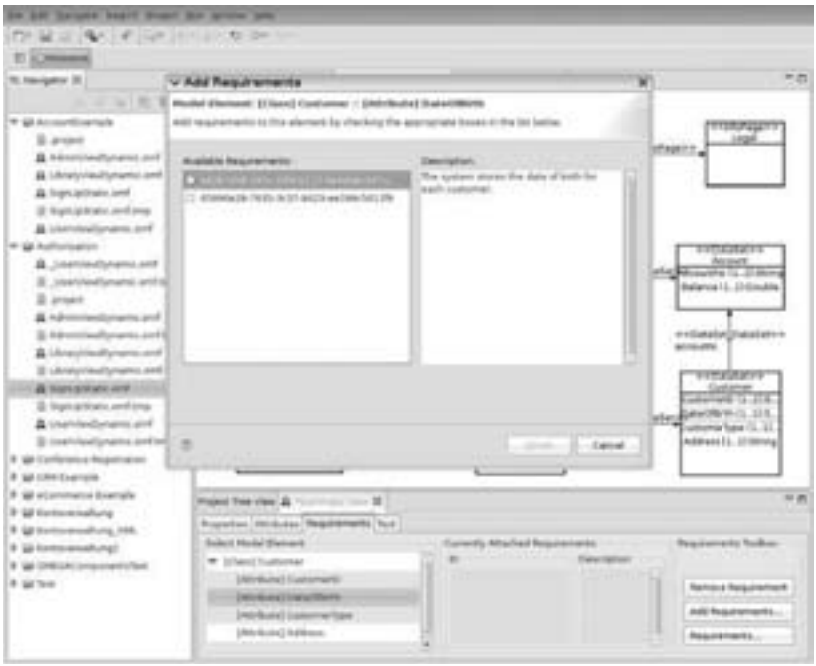

Figure 2: Creating traces between model elements and requirements

links from serialized models and source code artifacts. We have used our prototype in conjunction with the TraVis trace management and visualization tool (cf. [27]) and in the following sections will provide an overview on our findings.

4.4.3 Validation of Traceability Links

Our approach supports validation of traceability information in various ways. On a most basic level, links between model elements and related requirements can be validated automatically. Thus, using each requirement's unique identifier, requirements that no longer exist can be identified and references to these requirements can be deleted. Due to performance issues, a lazy validation and update policy was chosen for this kind of trace validation in our prototype. Therefore, references to requirements are validated each time a model element is accessed. Invalid references are then deleted automatically.

The correctness of links between model elements and related requirements has to be validated manually, however, since the modeller ist responsible for introducing references from requirements to model elements (cf. section 3.2). Consequently, the modeler also must make sure that the references that have been introduced remain valid. To facilitate the discovery of broken traceability links, trace visualization and management facilities can be employed as described in the following section.

4.5 Trace Management and Visualization

In order to support model-driven development efforts in locally distributed scenarios, we suggest the use of a *collaborative software development platform* (CSDP), that supports the management of all artifacts of a development project and provides controlled access to these resources to the stakeholders involved.

Additionally, the CSDP contains different synchronous and asynchronous communication means as well as an embedded wiki engine to facilitate collaborative documentation and information sharing. In our approach, requirements, in the form of issue tracker items, models of different levels of abstraction as well as the source code of an application are managed within a CSDP. Every artifact is supplied with a unique identifier, which also enables the built-in wiki engine to refer to these, e.g., when writing the documentation for future maintenance (see [27] and [29] for concrete features of the CodeBeamer CSDP).

Thus, the associations among elements can be managed as CSDP hyperlinks. Moreover, we have developed a complementary tool for trace visualization (TraVis) that is tailored at supporting traceability management even in distributed MDSD settings (see figure 3). Commonly used requirements management tools, on the other hand, do not provide full end-to-end traceability, whereas commonly available collaboration platforms lack integrated support for collaborative requirements elicitation, specification, and management processes (cf. [27] for a more detailed demarcation of existing traceability management tools).

Figure 3: Visualizing traces between models, requirements, and tracker items

Traces among requirements and different models can be extracted from the CSDP and project managers or developers are enabled to manage the traceability information visually, i.e., discover and repair inconsistencies. In addition, TraVis provides functionality for

conducting impact analyses starting from one particular artifact, e.g., a requirement or model, and exploring those artifacts affected when changing something at the point of origin. This in turn allows for detailed calculations on the estimated entailing costs of a particular change and thus its economic feasibility [37].

5 Contributions and Limitations

In the previous sections we have described the set of tools that support our approach to tracing requirements across different stages of artifact elaboration in model-driven development processes. The main contribution of our approach consists in revealing and proposing a prototypical solution for maintaining end-to-end traceability in distributed model-driven development processes. In doing so, a continuous and integrated tool chain for supporting software modeling and code generation (OMEGA), trace capturing and validation (CodeBeamer), well as trace information management and visualization (TraVis) is presented.

Moreover, the basic applicability of the approach has been shown by implementing the tool chain mentioned above based on both existing development tools and integrated custom-built prototypes that have already been evaluated in other related contexts (see [27]).

Future extensions to the prototype will include tools for parsing models and code. These will provide the means to perform a number of tasks, such as automatically deriving implicit traceability links from the explicit links (cf. section 3.3), the validation of these links, and the interpretation and visualization of the traceability information. TraVis is currently evaluated and further adapted to the TRACES approach for visualization of traces among model elements stored in a CSDP [37].

6 Discussion and Related Work

The general use of identifiers and references is a basic technique utilized in a wide range of application domains, so it will not be discussed further, here. In this section, we will provide an overview of related research efforts in the field of requirements traceability in order to justify the TRACES approach and highlight its unique features as opposed to existing solutions. This way, an evaluation of the approach's unique utility is conducted descriptively and based on the current body of knowledge in software engineering and information systems research [26].

There are many instances in literature describing automatic or semi-automatic approaches to trace capturing and management. For example, Antoniol et al. have developed a number of approaches based on information retrieval (IR) techniques [4, 2, 6, 3] and on the observation of test scenarios [7] that are largely supported by tools automating the process. Similar research on applying IR for candidate link generation has been conducted by Huffman-Hayes et al. [28]. However, these approaches inherently do neither support

MDSD processes nor distributed collaboration processes. TRACES, on the other hand, was purposefully designed to support this class of software engineering problems.

Egyed also suggests the use of test scenarios in order to obtain traceability information [19, 20, 21]. This approach can only be partly automated, though, due to the fact that traces need to be established and managed mainly manually. Spanoudakis et al., on the other hand, describe a rule-based approach which can almost completely be automated [38]. So do Jirapanthong and Zisman, who develop another rule-based approach to the automatic generation of traceability links [30]. Since model-to-model and model-to-code transformations also contain rules, these approaches are somewhat comparable to ours. However, as it is the case with the IR approaches, none of the rule-based approaches can be smoothly integrated with existing MDSD and CSDP tools in order to support distributed MDSD.

There are only a few authors emphasizing the importance of tool integration with respect to traceability. Even though some tool support for traceability, i.e., trace capturing, management, and analysis exists, Aizenbud-Reshef et al. conclude that there is a lack of integrated solutions [1]. Complementarily, Kowalczykiewicz and Weiss stress the importance of tool integration [32]. They present a first prototype for an integrated tool platform with support for traceability and change management whereas there are no particular capabilities for MDSD yet.

Therefore, the unique utility of the OMEGA TRACES approach consists of allowing a relatively high degree of automation with respect to trace generation and capturing traces among requirements and model elements at different levels of abstraction due to a broad information basis stored centrally within the collaboration platform. Moreover, TRACES is designed to integrate with existing development environments and collaboration platforms in order to support collaboration in distributed model-driven development processes commonly found in outsourced and offshore software projects (cp. [37]). In addition, the integration with TraVis accounts for visual traceability and change management in distributed projects (see section 4.5). As has been shown, both analytically and empirically, in [27], collaborative and visual traceability management accounts for superior productivity as compared to commonly available solutions. Therefore, this custom-developed solution is chosen here in combination with the TRACES prototype.

7 Conclusion

We have presented an easy to use, yet inherently powerful approach to traceability and a prototypical implementation, which allows a high level of automation, tool integration, and distributed collaboration. By integrating our prototype with the Eclipse and CodeBeamer platforms, we have demonstrated that the approach can easily be used in conjunction with widely accepted development environments for realizing locally distributed MDSD projects. Moreover, our prototype supports automated generation, validation, and update of traceability information.

A main advantage of our approach is the ease of use and therefore easy adoption for devel-

opers, since little additional effort is necessary to create a reasonable amount of traceability information that can be retrieved from models and code using external tools, such as TraVis. Due to the fact that all information needed to link requirements to model elements and source code and other textual resources is stored in XML/XMI-serialized models, our approach can easily be adapted to other modeling and code generation environments that use these standards.

There are, however, a few minor issues that will be addressed in the near future. Most significantly, the current strategy of validating and updating traceability links allows "invalid" model states where model elements reference requirements that no longer exist. This is because a lazy validation and update policy was chosen for the prototype to improve its performance. However, a more efficient solution for this particular problem will be developed and evaluated empirically in case study and/or experimental settings.

References

[1] Aizenbud-Reshef, N., Nolan, B. T., Rubin, J. and Shaham-Gafni, Y.: Model Traceability. IBM Systems Journal, **45** (2006) 515-526

[2] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A. and Merlo, E.: Tracing Object-Oriented Code into Functional Requirements. Proceedings of the IEEE International Workshop on Program Comprehension (IWPC), Limerick (June 2000)

[3] Antoniol, G., Canfora, G., Casazza, G. and De Lucia, A.: Information Retrieval Models for Recovering Traceability Links between Code and Documentation. Proceedings of the IEEE International Conference on Software Maintenance (ICSM), San Jose (October 2000)

[4] Antoniol, G., Canfora, G., De Lucia, A. and Merlo, E.: Recovering Code to Documentation Links in OO Systems. Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE), Atlanta, Georgia (October 1999)

[5] Antoniol, G., Caprile, B., Potrich, A., and Tonella, P.: Design-code Traceability Recovery: Selecting the Basic Linkage Properties. Science of Computer Programming, **40** 2001, 213-234

[6] Antoniol, G., Casazza, C. and Cimitile, A. Traceability Recovery by Modeling Programmer Behavior. Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE), Brisbane (November 2000)

[7] Antoniol, G., Merlo, E., Gueheneuc, Y.-G. and Sahraoui, H.: On Feature Traceability in Object Oriented Programs. Proceedings of the International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), Long Beach (November 2005)

[8] Apache Software Foundation: The Apache Velocity Project. `http://velocity.apache.org` (2007)

[9] Atkinson, C., Kühne, T.: The Role of Metamodeling in Model-Driven Development. International Workshop in Software Model Engineering (held in conjunction with UML '02), Dresden (October 2002)

[10] Bézivin, J.: MDA: From Hype to Hope, and Reality. Invited Talk at UML 2003, San Francisco (October 2003)

[11] Booch, G., Brown, A., Iyengar, S., Selic, B.: An MDA Manifesto. MDA Journal, 2004

[12] Brown, A.: Model driven architecture: Principles and practice, Software and Systems Modeling, Volume 3, Number 4 (December 2004), pp. 314-327, Springer, Berlin, Heidelberg, 2004

[13] Codehaus: XStream. http://xstream.codehaus.org(2007)

[14] CMMI Product Team: CMMI for Systems Engineering/Software Engineering/Integrated Product and Process Development/Supplier Sourcing, Version 1.1 Carnegie Mellon Software Engineering Institute, 2002

[15] Czarnecki, K. and Helsen, S.: Classification of Model Trans-formation Approaches. 2nd OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture, Anaheim, CA, 2003.

[16] Domges, R. and Pohl, K.: Adapting Traceability Environments to Project-Specific Needs Communications of the ACM, **41** (1998) 12, pp. 54-62

[17] Eclipse Foundation: Eclipse Development Platform. http://www.eclipse.org (2007)

[18] Eclipse Foundation: Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/ (2007)

[19] Egyed, A.: A Scenario-Driven Approach to Traceability. Proceedings of the International Conference on Software Engineering (ICSE), Toronto (May 2001)

[20] Egyed, A.: A Scenario-Driven Approach to Trace Dependency Analysis. IEEE Transactions on Software Engineering (TSE), **29** (2003) 2, pp. 116–132

[21] Egyed, A.: Resolving Uncertainties during Trace Analysis. Proceedings of the 12th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), Irvine, CA (November 2004)

[22] Gitzel, R.: Model-Driven Software Development Using a Metamodel Based Extension Mechanism for UML. Peter Lang Europäischer Verlag der Wissenschaften, Frankfurt (2006)

[23] Gitzel, R., Ott, I., Schader, M.: Ontological Extension to the MOF Metamodel as a Basis for Code Generation, in: The Computer Journal, 50 (2007) 1, pp. 93–115

[24] Gitzel, R., Schwind, M.: Using Non-linear Metamodel Hierarchies for the Rapid Development of Domain-Specific MDD Tools. Proceedings of Software Engineering Applications (SEA), Dallas, Texas (November 2006)

[25] Glass, R.L.: The Generalization of an Application Domain. IEEE Software, **17** (2000) 5, pp. 127–128

[26] Hevner, A. R., March, S. T., Park, J., Ram, S.: Design Science Information Systems Research MIS Quarterly, MISQ Discovery, **28** (2004) 1, pp. 75–105

[27] Hildenbrand, T.: Improving Traceability in Distributed Collaborative Software Development – A Design Science Approach. Dissertation Thesis, University of Mannheim, Germany (2008)

[28] Huffman Hayes, J., Dekhtyar, A., Sundaram, S. K.: Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. IEEE Transactions on Software Engineering, 32 (2006) 1, pp. 4–19

[29] Intland Software: Collaborative Software Development Solution. http://www.intland.com/products/codebeamer.html (2007)

[30] Jirapanthong, W. and Zisman, A.: Supporting Product Line Development through Traceability. Proceedings of the Asia-Pacific Software Engineering Conference (APSEC), Taipei (December 2005)

[31] Kleppe, A., Warmer, J. and Bast, W.: MDA Explained - The Model Driven Architecture : Practice and Promise. Addison-Wesley, 2003.

[32] Kowalczykiewicz, K. and Weiss, D.: Traceability: Taming uncontrolled change in software development. Proceedings of the National Conference on Software Engineering, Tarnowo Podgórne, Poland (2002)

[33] Lindvall, M., Sandahl, K.: Practical Implications of Traceability. Software – Practice & Expererience, **26** (1996) 10, pp. 1161-1180

[34] Mellor, S., Balcer, M.: Executable UML - A Foundation for Model-Driven Architecture. Addison-Wesley, Hoboken, 2000.

[35] Miller, J., Mukerji, J.: MDA Guide Version 1.0.1, OMG Document Number: omg/2003-06-01, OMG, `http://www.omg.org/cgibin/doc?omg/2003-06-01`

[36] Redmiles, D., van der Hoek, A., Al-Ani, B., Hildenbrand, T., Quirk, S., Sarma, A., Filho, R. S. S., de Souza, C.R.B. and Trainer, E.: Continuous Coordination: A New Paradigm to Support Globally Distributed Software Development Projects WIRTSCHAFTSINFORMATIK, 49 (2007) Special Issue, pp. S28-S38

[37] de Souza, C.R.B., Hildenbrand, T. and Redmiles, D.: Towards Visualization and Analysis of Traceability Relationships in Distributed and Offshore Software Development Projects. Proceedings of the First International Conference on Software Engineering Approaches For Offshore and Outsourced Development (SEAFOOD'07), Springer Lecture Notes in Computer Science (LNCS), Zurich, Switzerland (February 2007).

[38] Spanoudakis, G., Zisman, A., Perez-Minana, E. and Krause, P.: Rule-based generation of requirements traceability relations. Journal of Systems and Software, **72** (2004) 2, pp. 105–127

[39] Stahl, T., Voelter, M.: Model-Driven Development: Technology, Engineering, Development. Wiley, 2006

[40] Vanhooff, B. and Berbers, Y.: Breaking Up the Transformation Chain. 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), Workshop on Best Practices for Model Driven Software Development, San Diego (October 2005)