Model traceability

N. Aizenbud-Reshef
B. T. Nolan
J. Rubin
Y. Shaham-Gafni

Traceability relationships help stakeholders understand the many associations and dependencies that exist among software artifacts created during a software development project. The extent of traceability practice is viewed as a measure of system quality and process maturity and is mandated by many standards. This paper introduces model traceability, reviews the current state of the art, and highlights open problems. One issue that impedes wide adoption of traceability is the overhead incurred in manually creating and maintaining relationships. We review the latest research advancements that address this issue through the automatic discovery of trace relationships. Model-driven development provides new opportunities for establishing and using traceability information. We discuss automatic generation of trace information through transformations and the use of traceability relationships to maintain consistency and synchronize model artifacts. We conclude with a discussion of the implementation and utilization challenges that lie ahead.

INTRODUCTION

Models are used in software development to manage complexity and communicate information to many stakeholders. There are models for business processes, system requirements, architecture, design, and tests. Each model has its own notation, representation, tools, and users. Thus developers, tools, artifacts, and processes are largely isolated and only weakly integrated. Interconnections are largely implicit, opening the door for inconsistencies and making it difficult to propagate change. End-to-end integration can make these relationships explicit and maintain traceability information throughout.

Model-driven development (MDD) provides an opportunity to automate both the creation and discovery of traceability relationships, and to maintain consistency among the heterogeneous models used throughout the system-development life cycle. The formality or semiformality of models makes it possible to apply analysis methods, which may then serve as a basis to automate traceability. In addition, model transformations (forward, like code generation, or backward, like reverse engineering) can be the source of generated links or mappings.

OVERVIEW

The IEEE Standard Glossary of Software Engineering Terminology[1] defines traceability as follows:

(1) The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match; (2) The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement that it satisfies.

This definition is strongly influenced by the originators of traceability—the requirements management community. Gotel and Finkelstein[2] define requirements traceability as follows:

. . . the ability to describe and follow the life of a requirement, in both a forward and backward direction; i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases.

We suggest a much broader definition of traceability. We regard traceability as any relationship that exists between artifacts involved in the software-engineering life cycle. This definition includes, but is not limited to the following:

- Explicit links or mappings that are generated as a result of transformations, both forward (e.g., code generation) and backward (e.g., reverse engineering)
- Links that are computed based on existing information (e.g., code dependency analysis)
- Statistically inferred links, which are links that are computed based on history provided by change management systems on items that were changed together as a result of one change request

Traceability is achieved by defining and maintaining relationships between artifacts involved in the software-engineering life cycle during system development. In this paper, we use the words *relationship* and *link* interchangeably to denote a traceability relationship.

Traceability is mandated by many standards, such as MIL-STD-498, IEEE/EIA 12207, and ISO/IEC 12207. These standards derive from the waterfall methodology in which the role of traceability stemmed from the need to show that the resulting system met contractual agreements. Such standards reflect the view that traceability practice is a measure of system quality and software process maturity. This view is illustrated by the Software Engineering Institute's concept of Capability Maturity Model**.[3]

Different stakeholders in the software development process have different traceability goals. The project manager[4] perspective is that traceability supports demonstrating that each requirement has been satisfied and that each system component satisfies a requirement. From the perspective of requirements management, traceability facilitates linking requirements to their sources and rationales, capturing the information necessary to understand the evolution of requirements, and verifying that the requirements have been met. During design, traceability enables designers and maintainers to keep track of what happens when a change request is implemented before a system is redesigned.[5] With complete traceability, more accurate costs and schedules of changes can be determined, rather than depending on the programmer to know all the areas that will be affected by these changes.

Although the advantages are well documented, traceability practice is not widespread. The commonly stated reason is the high cost of manual creation and maintenance of traceability information.[6] In addition, the lack of guidance as to what link information should be produced and the fact that those who use traceability are commonly not those producing it also diminishes the motivation of those who create and maintain traceability information.

The development and use of techniques to trace requirements originated in the early 1970s to provide the means to answer a range of questions, such as: Is this requirement necessary? Why is the design implemented this way, and what were the other alternatives? What is the impact of changing a requirement?

The first method used to express and maintain traceability was cross-referencing. This involves embedding phrases like "see section x" throughout the project documentation. Since those days, many different techniques have been used to represent traceability relationships including standard approaches such as matrices,[7,8] databases,[9] hypertext links,[10] graph-based approaches,[11] formal methods,[12] and dynamic schemes.[13]

Automated support for traceability began with general-purpose tools such as word processors, spreadsheets, or database systems and became easier to use with the advent of hypertext technology. Still, the major drawback of this method remains: The traceability information is created and maintained manually, as is the responsibility for managing its validity with respect to change. Thus, the traceability information quickly becomes outdated as the system evolves.

Special-purpose requirements management tools, such as IBM RequisitePro*[14] and Telelogic DOORS**,[15] introduced more advanced traceability solutions, which support the management of traceability information validity by monitoring changes of linked elements and indicating suspect links. They also support integration with other software development tools to facilitate traceability from requirements to other products of the software life cycle. Despite these advances, the burden of keeping traceability information current remains a manual task because requirements, most commonly expressed as informal text, require a human to understand and determine link validity. In addition, most tools do not provide rich traceability schemes, thus allowing only simple forms of reasoning about traces.

STATE OF THE ART

In this section, we review the state of the art of traceability technology and methodology and the potential that MDD brings to the field. We discuss technologies to implement traceability, the latest advancements in the automatic discovery of trace relationships, and the complexities of managing traceability relationships as software artifacts evolve.

Technology

When building traceability support into their tools tool developers face several challenges. Some of the major issues stem from the need to reference software artifacts that are external to the tool. We next discuss issues in the representation, persistence, and maintenance of traceability relationships.

Metamodel

The most basic traceability solution provides the ability to link artifacts but does not provide semantics for these relationships; the link merely represents the fact that there is some relationship between the artifacts and allows the user to trace from one to the other. Many papers discuss the need to distinguish between different types of relationships with specific semantics in order to facilitate and support richer analysis and reasoning about

■ Model-driven development provides new opportunities for establishing and using traceability information ■

traces. There are two approaches to supporting richer semantics. The first is to allow users to add attributes to relationships; they can define relationship types that are aligned with their process and the types of artifacts they use. Then they can perform queries based on those attributes. This approach is taken by some commercial tools such as DOORS, but its limitation is that the tool treats all attributes uniformly and cannot provide specialized behavior for different types of relationships.

The second approach is to provide a predefined standard set of relationship types that can be supported by the tool. This approach is presented by Ramesh and Jarke.[16] They made extensive observations of traceability practice in several organizations, and their analysis revealed that participants fell into two distinct groups, which the authors refer to as *low-end* and *high-end*. Low-end users view traceability simply as a mandate from project sponsors and use simple traceability schemes to model dependencies among requirements, system components, and compliance verification procedures. High-end users view traceability as an important component of a quality systems engineering process and employ much richer traceability schemes, thereby enabling more precise reasoning about traces.

The authors propose two levels of reference models. The first level, aimed at low-end users, provides a handful of relationship types. The second level provides a richer set of link types for high-end users. Some examples of relationship types are *satisfies,* *justifies, describes, depends on*, and *validates*. Additional authors have also proposed traceability metamodels.[11,17,18] The limitation of this approach is that the types of relationships are fixed, while organizational needs and practices vary. We believe an optimal solution should provide a predefined link metamodel, allow customization, and allow extensibility to define new link types.

Linkage information

Most existing traceability solutions are provided by requirement management tools and support tracing requirements to other products of the software life cycle. Examples of such commercial tools are DOORS, Tcp Sistemas e Ingeniería IRqA**, Igatech Systems RDT, Reqtify, IBM RequisitePro,[14] and Verocel VeroTrace**.[19] Typically, these tools store link information within the artifacts themselves. This limits the types of related artifacts to only those the tool can recognize and manipulate. Links to artifacts external to the tool are achieved by specific integration with other tools or by treating all artifacts as text artifacts and making the user responsible for understanding artifact semantics and knowing where to position links in the text. Keeping the link information with the artifacts is problematic for several reasons: Each artifact has its own representation and semantics, which makes performing an analysis difficult; each new type of artifact or tool requires a special integration effort; and if the link is directed and stored only in the source artifact, it is not visible from the target artifact. If link information is stored in both artifacts, it adds to the burden of maintaining consistency when changes to the link are made.

MDD places new demands on such traceability tools, which must now be capable of dealing with different types of models—such as business, data, design, and test—and their artifacts. In addition, the need to support collaborative and multisite development adds new types of artifacts, such as team-room documents, chat discussions, and e-mails.

In one approach to this problem all artifacts are kept in a *metadata repository*.[20] This solution, although attractive from the traceability point of view, is less appealing to tool vendors because it requires tight integration and strong coupling. A compromise leaves the responsibility for the management of artifacts to the tool while keeping some replicated artifact information in a shared repository. Relationships are kept either as properties in the representation of artifacts in the repository or as "first-class citizens," having their own representation in the repository. Examples of solutions that follow this approach are given in References 11, 21, and 22.

Keeping link information separate from the artifacts requires the ability to uniquely identify artifacts across space and time. Tool artifacts may not always have a unique identifier, especially if their granularity is smaller than physically stored artifacts (e.g., a method in a Java** class). Technologies such as link anchors and bookmarks can be used to identify such artifacts, but more research is required to make such anchors robust when artifacts are edited, cut, and pasted.

Automated creation

The increased burden of manually specifying and maintaining traceability information is a major impediment to the general acceptance of traceability practice.[6] Much of the recent research has focused on finding ways to automate both the creation and maintenance of traceability information. One research direction is to employ text mining and information retrieval techniques to infer relationships between artifacts. Alexander[10] describes a semiautomatic method for creating links between use cases and use-case references and between terms and their definitions in a glossary. Markup is used to mark the terms that users are interested in linking. Automatic tools then search for the related artifact and generate hypertext documents. Huffman Hayes et al.[23] have studied a method for improving candidate link generation by applying information retrieval techniques. They focus on improving recall and precision to reduce the number of missed traceability links and to reduce the number of irrelevant potential links that an analyst has to examine when performing requirements tracing. Several information retrieval algorithms were evaluated by comparing their results to those of a senior analyst who traced both manually and with an existing requirements tracing tool. Initial results suggest that information-retrieval methods can retrieve a significantly higher percentage of the links than analysts can, even when they use existing tools, and do so in much less time while achieving comparable ratios of meaningful to irrelevant links. Antoniol et al.[24] propose a method based on information retrieval to recover traceability links between source code and free text documents. They apply both a probabilistic and a vector space information retrieval model in two case studies to trace C++ source code onto manual pages and Java code to functional requirements. Marcus and Maletic[25] use latent semantic analysis, an advanced information retrieval technique, to extract the meaning (semantics) of the documentation and source code, and then use this information to identify traceability links based on similarity measures.

Another research direction involves the analysis of existing relationships to obtain implied relations. For example, the work of Sherba et al.[22] builds on techniques from open hypermedia and information integration. These techniques provide generic services that enable the discovery, creation, maintenance, viewing, and traversal of relationships. Users of this system can define new derived relationships as a chain of existing relationship types ($\mathrm{rel}_1 \rightarrow \mathrm{rel}_2 \rightarrow \ldots \rightarrow \mathrm{rel}_n$), and the system automatically discovers instances of the derived relationships. Eyged and Grünbacher[26] present an approach that requires minimal initial trace information between scenarios and the tests that validate them and between tests and design elements. Trace dependencies between requirements and code can be automatically inferred by checking which code classes were activated as a result of running a test scenario for a specific requirement. Once this information is available, analysis methods are applied to find additional trace dependencies among requirements and between requirements and model elements (e.g., state transitions).

Analyzing change history may also provide a source of automatically computed links. Ying et al.[27] and Zimmermann et al.[28] apply different data mining techniques to the change history of a code base to determine change patterns, that is, sets of files that were changed together frequently in the past. Files in the same change set are related and can be used to recommend potentially relevant source code to a developer performing a modification task.

Relationship management

One of the most challenging aspects of traceability is how to maintain the correctness and relevance of relationships while the artifacts continue to change and evolve. This is especially critical for links that are maintained manually because the traceability process is unclear and an imbalance between the amount of work involved and the perceived benefits often results in insufficient resources being devoted to traceability maintenance. Thus, traceability information gradually erodes.

The manner in which a relationship is managed depends on its nature and origin. Both manual effort and computation time need to be invested to manage relationships. The goal is to minimize manual effort at the expense of increasing computation resources, but this approach is a bit simplistic. The number of relationships may be huge and the computation solution may not scale. The following definitions can help clarify relationship management trade-offs.

An *imposed* relationship is a relationship between artifacts that exists by volition of the relationship creator. For example, a *verifies* relationship can be created between a test and a requirement. The relationship exists until the creator decides to remove it. When the requirement changes, it is not

■ A well-defined, automated method of accomplishing traceability would be of value in any domain, on any project, with any methodology ■

clear if the relationship is still valid; thus, it becomes *suspect*, and the tester can check the test and determine if the relationship is still valid or if the test no longer verifies the requirement, in which case the relationship becomes *invalid*.

An *inferred* relationship between artifacts is one that exists because the artifacts satisfy a rule that describes the relationship. This inference can be done either automatically by a computer or manually. For example, if method `m1` calls method `m2`, then there is a call-dependency relationship between `m1` and `m2`. An inferred relationship cannot be *invalid*, because if the rule is not satisfied, the relationship simply does not exist. An inferred relationship is *suspect* if one of its related artifacts has changed and it is not clear if the rule is still satisfied. To determine if the relationship still exists, some action needs to be taken to reestablish the relationship existence.

A *manual* relationship is a relationship that is created and maintained by a human user. This relationship can be either imposed or inferred. If a manually related artifact changes, the relationship becomes *suspect*, and a person needs to check if the relationship still exists or to reestablish its validity.

A *computed* relationship is one created and maintained automatically by a computer. There are two basic types of computed relationships: *derivation* and *analysis*.

A *derivation* relationship denotes that, given the content of one artifact, it is possible to compute valid content for the related artifact. This kind of relationship is typically created when code generation or model transformations are run. A derivation relationship is a type of imposed relationship that is valid at the time of generation. If a source artifact is later changed, the relationship becomes *suspect* and requires reapplying the derivation (transformation or code generation) to assure its validity. If a derived (target) artifact is changed, again the relationship becomes *suspect*. In this case, the course of action depends on the directionality of the relationship. (Unidirectional transformations can be executed in one direction only, in which case a target model is computed or updated, based on a source model. Bidirectional transformations can be executed in both directions, which is useful in the context of synchronization between models.[29]) If the relationship is directed, then there needs to be a check whether the changed artifact still satisfies the derivation relationship, and the relationship is either valid or invalid according to the result. If the relationship is bidirectional, then we assume the existence of a reverse derivation function that is applied from the target to the source and can change the source so that the derivation relationship is again valid.

An *analysis* relationship is a type of inferred relationship created by analysis programs that examine code or models against a set of rules. Discovering dependency relationships between a Java class and the classes it depends on through import is an example. Typically, analysis programs are computation-intensive. Because a change in one of the related artifacts makes the relationship suspect, the analysis program needs to be rerun. If there are many such relationships, a small change can trigger a huge amount of computation. Thus, in these kinds of relationships it makes sense to use lazy computation. The analysis is reapplied only when a relationship is needed but has been marked *suspect*.

Several researchers have proposed ways to improve the maintenance of manual relationships. Huang-Cleland et al.[30] propose event-based traceability, in which requirements and other software engineering artifacts are linked through publish-subscribe relationships: Requirements and other change instigators take on the role of publishers, and dependent artifacts act as subscribers. When requirements are changed, events are published to an event server and related notifications sent to all dependent subscribers. Messages carry sufficient information to provide meaningful semantics about each event to support the update process. Olsson and Grundy[31] describe an approach in which all changes are tracked. Some changes can be resolved automatically, and for the rest, developers are informed so that they can take appropriate action.

Marcus and Maletic[25] propose a way to deal with relationship validity in an automated way by defining a *validity scale*. They define the notion of causal conformance relationships that represent logical semantic dependencies among documents with an implied logical ordering over time. These relationships are represented by using a hypertext model, and a conformance analysis method compares node-modification time stamps and link-validation time stamps to produce a heuristic conformance rating of the likely seriousness of conformance problems.

Freude and Königs[32] propose an approach for automating consistency management among related artifacts using reference objects and consistency relations. Each element in any of the models is referenced by a reference object. Consistency relations are defined between reference objects and provide conditions that must be fulfilled. The relations can be used to navigate between referenced objects, detect consistency violations, and maintain consistency.

The introduction of configuration management makes relationship management more complicated. In addition to relationship validity, relationship versions need to be managed. Watson[33] describes the complex issues of managing requirements and respective traceability relationships across versions and their connection to project milestones.

Methodology

The system development methodology can significantly impact the ease or difficulty of producing traceability artifacts and performing analyses. All methodologies describe what work products need to be produced, and many include a traceability matrix as one of their work products. Usually no guidance is given as to how this should be produced, and most end up being produced manually, with the attendant costs and issues.

One can create a system model that includes a requirements metamodel. In fact, this is the intent of Systems Modeling Language. The requirements metamodel and the resulting model that includes a requirements model explicitly provide for traceability from the requirements to the other artifacts that implement or realize them. The existence of such a model, however, again provides no guidance on how to produce (preferably automatically) the trace relationships between the requirements and their realization. In other words, the fact that we are able semantically to produce meaningful trace relationships gets us no further along the road to an effective traceability strategy. The relationships still need to be built and maintained manually. It would be better if we could generate the artifacts, or at least the skeleton of them, produce the traceability links as we generate them, and then fill in details as needed. As discussed below, Model Driven Architecture** (MDA**) strategies provide the promise of being able to do this.

The utilization of a use-case-centric methodology, such as the IBM Rational Unified Process*,[34] can facilitate traceability and provide an effective basis for an MDA. Use cases, a concept developed by Jacobson et al.,[35] provide a unifying and often simplifying concept for a system and can drive development, testing, and documentation efforts.

A *use case* is defined as a sequence of actions that returns value back to an actor. The collection of use cases that represents all the significant interactions between the system and the outside world (between the system and its actors) constitutes the system-level functional requirements. Other important requirements exist, but they can be expressed as either constraints on the whole system or constraints on the system use cases. This distinction is not meant to devalue other requirements, for very often these constraints and requirements determine the very nature of the system. It is a matter of focus: By concentrating on the use cases, one can provide context for nonfunctional requirements and can organize them around the use cases. This facilitates traceability. The nonfunctional requirements can be traced to the use cases, but this is likely to be manual, although it could be facilitated by some of the techniques discussed earlier.

Each use case is a thread through the system that returns significant value back to an actor or the domain in which it exists. It can be thought of as a horizontal slice through the system, which usually exercises architecturally significant portions of the system. Development efforts can therefore be organized around the use cases. Each completed use case provides significant value, facilitating iterative development, reducing risk, and enhancing stakeholder satisfaction.

Because a use case, by definition, returns a value, that value can be tested. Testing can be organized by use cases, and the test cases can be associated with the use cases. This continues to simplify the organization of artifacts and provides another dimension of traceability. This traceability can be done either manually, with its drawbacks, semi-automatically by using a naming convention and a script or program to generate traceability information, or automatically by generating a test-case skeleton from the use case with a transformation. Once a use case has been defined, at least the skeleton of a test case for it should be generated automatically.

The IBM Rational* Unified Process for Systems Engineering (RUP SE),[36] a variant of the RUP*, was developed to address the needs of large-scale systems and the systems engineers who develop them. The RUP SE facilitates traceability through a technique called *use-case flowdown*, based on a logical decomposition of the system. The RUP SE activity of architectural analysis posits a set of subsystems that collaborate to fulfill the responsibilities of the system. Use-case flowdown describes the collaborations among subsystems and derives subsystem requirements through those collaborations. Each subsystem can then be considered a system in itself, and use-case flowdown can be recursively applied.

A use-case realization describes how a set of entities collaborate to "realize" a use case. Use cases may be realized through three types of models: collaboration of subsystems and classes, state machines, or activity models. Again, once a use case is identified, a use-case realization for it can be generated using a transformation. The use case can be traced to the realization, either manually or preferably automatically, through the transformation. The realization contains a set of participants, which can therefore be traced to the use case (requirement). In some cases, this level of traceability is sufficient. In other cases, it is necessary to trace further; each participant is considered a system in its own right, and the process is recursively applied.

Each participant in the realization requires information or services from the other participants. In an object-oriented paradigm, they obtain these services by sending messages to the other participants by calling the operations of the other participants.

■ An optimal solution should provide a predefined link metamodel, allow customization, and allow extensibility to define new link types ■

These operations are analogous to finer-grained use cases in the context of the participants; there are certain requirements imposed on them, and it is expected that values will be returned. These use cases or operations will have their own realizations with a different set of participants.

This chain of use case → use case realization → participant → operation provides a basis for traceability through an arbitrary number of levels of decomposition. In effect, by following a rigorous development process, a logical call tree can be developed. A significant part of system traceability can be obtained by tracing through the call tree. It is very difficult to trace this manually, less difficult to do so if an appropriate naming scheme is applied and a traceability script or program written, and easier yet if traces are generated by transformations.

Traceability and MDD

The model-driven approach focuses on models as the primary artifacts. MDD recognizes the necessity of having several kinds of models to represent the system as it progresses from early requirements through final implementation. These models may represent different aspects of the system (e.g., structural or behavioral), or they may represent the system at varying levels of abstraction (e.g., an analysis model or a design model). In this section, we discuss the role of traceability in maintaining consistency among models and in model transformations.

Inconsistencies among models representing different viewpoints of a system, or among specifications at differing levels of abstraction, can arise during or between phases of software development, raising the compelling issue of how to manage consistency among different models and between models and code.[37] Grundy et al.[38] provide an excellent review of various tools and different approaches to inconsistency management. According to their view, some inconsistencies may be automatically corrected; however, many inconsistencies cannot, or should not, be automatically corrected. Hence, mechanisms are required to inform developers of inconsistencies and facilitate the monitoring and resolution of inconsistencies.

Desfray believes that traceability is an essential part of the solution.[39] Several recent papers have proposed to use traceability as the basis for detecting and informing related stakeholders about inconsistencies.[30,31,40] As it is usually impossible to keep a software system consistent at all times, tools need to have different policies for consistency enforcement. Future work in this area is needed to define a traceability model that supports detecting, representing, storing, and propagating a wide range of inconsistencies, and tying these with appropriate inconsistency management policies.

MDA,[41] the Object Management Group approach to MDD, is based on modeling and automated mapping of models to implementations through model transformation. Two standardization efforts are underway to realize model transformations: the MetaObject Facility (MOF**) 2.0 Query/Views/Transformations (QVT)[42] and the MOF Model-to-Text Transformation Language.[43] Transformations may be unidirectional or bidirectional. Updating unidirectional transformations and synchronizing bidirectional transformations require the means to identify the existing target model element related to a given source model element. Thus, such transformations need to create and maintain mappings between source and target models.

Technologies to perform model transformations range from conventional programming languages to specific transformation languages.[44] Czarnecki and Heslen[29] propose a taxonomy for the classification of model transformation approaches. The generation of traceability links is one of the dimensions of this classification. According to Czarnecki and Heslen, transformation approaches divide into those that provide dedicated support for traceability and those that depend on the user to encode the generation of traceability, using the same mechanisms as those for generating any other kinds of model elements. In the case of automated support, the approach may still provide some control over how many traceability links are created (in order to limit the amount of traceability data). Finally, there is the choice of the location at which the links are stored, for example, in the source and target, or separately.

Jouault[45] distinguishes between internal traceability, which is maintained by the transformation engine during the transformation to assist with the transformation algorithm, and external traceability, which is a persistent mapping kept beyond the transformation execution. Many approaches allow the serialization of internal traceability information. The main drawback is that the format and granularity in which such traceability is stored are predefined and may not be compatible with the representations used by other tools that make use of traceability information.

The QVT standard from OMG aims to provide a general-purpose language for achieving model transformations. The second revised submission of the QVT-Merge Group defines a hybrid declarative/imperative transformation language, with the declarative part being split into a two-level architecture: a high-level relational language and a low-level core language, where the relational language is compiled into the core language.[46] Traceability relationships between model elements involved in a transformation are created implicitly for the relational language and are defined explicitly as trace classes in the core language. Transformations have been designed to support incremental updating: Once a relationship (a set of trace instances) has been established between models by executing a transformation, small changes to a source model may be propagated to a target model by reexecuting the transformation in the context of the trace, causing only the relevant target model elements to be changed without modifying the rest of the model.

Although much progress has been done in the area of model transformations, the integration with traceability is still weak. Transformation solutions tend to define their own internal traceability and do not integrate well with existing traceability solutions that provide analysis and query capabilities on the traceability information.

CONCLUSION

We have reviewed the traceability landscape, with emphasis on issues relevant to the software life cycle and MDD. One of the main concerns is the lack of integration among the various environments and tools that need to share traceability information. A necessary step toward better integration is the definition of a standard traceability metamodel. Such a model should provide a definition of reference objects representing traceable artifacts and specify different types of traceability relationships with standardized semantics. The traceability metamodel should provide for customization and support extensibility mechanisms. In addition, a standard format for exchanging traceability information is needed.

Another issue that needs to be addressed is the ability to uniquely identify artifacts across space and time. Tools typically do not do this, but future tools will need to create and maintain globally unique identifiers for artifacts of different granularities.

Several approaches to automatically discover traceability relationships were discussed. These initial efforts are promising, yet they are still preliminary. More research is needed to discover additional and better techniques for automatic traceability creation. One approach that may be promising is to monitor user gestures to infer relationships. Each method for automatic relationship discovery provides one piece of the puzzle, but to obtain a complete picture, we still need to fit the pieces together and develop methods to integrate them to provide a complete solution.

We introduced the complexity and trade-offs involved in automated relationship management. Future work in this area is needed to define a traceability model that supports detecting, representing, storing, and propagating a wide range of inconsistencies, and tying these together with appropriate inconsistency management policies. One of the concepts introduced was a *validity scale*. Future investigation may help determine if the concept of varying degrees of confidence can help with prioritizing resource utilization (manual and computational).

The technology used to support traceability is only half the story. Equally, if not more important, is the relationship of traceability with the software development process and methodology. Traceability techniques originated in the 1970s when the waterfall methodology dominated. Since then, there have been many innovations in software development methodology, for example, agile and iterative processes. It may be interesting to revise traceability practice in light of these novelties.

Finally, we discussed the major role that traceability plays in model-driven engineering and the current state of the art. Even though much progress has been made, there are still many challenges that lie ahead. The integration of transformations and traceability needs improvement, possibly through a definition of a standard traceability metamodel.

ACKNOWLEDGMENTS

We thank Geoffrey Clemm for many interesting and enlightening e-mail discussions; his ideas served as the basis for the section on traceability relationship management. We thank Peri Tarr and Kim Letkeman for their insights and contribution to this paper, and also the anonymous reviewers whose comments helped us better focus the paper.

CITED REFERENCES

1. *IEEE Std 610.12-1990,* “IEEE Standard Glossary of Software Engineering Terminology,” © IEEE, New York (September 1990).
2. O. C. Z. Gotel and A. C. W. Finkelstein, “An Analysis of the Requirements Traceability Problem,” *Proceedings of the First International Conference on Requirements Engineering,* Utrecht, The Netherlands (1994), pp. 94–101.
3. Capability Maturity Model Integration (CMMI), http://www.sei.cmu.edu/cmmi/.
4. G. Stehle, “Requirements Traceability for Real-Time Systems,” *Proceedings of EuroCASE II,* London, England (April 1990), pp. 1–27.
5. M. Edwards and S. Howell, *A Methodology for System Requirements Specification and Traceability for Large Real-Time Complex Systems,* Technical Report, U.S. Naval Surface Warfare Center Daahlgren Division, Dahlgren, VA 22448 (1991).
6. P. Arkley, P. Mason, and S. Riddle, “Position Paper: Enabling Traceability,” *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering,* Edinburgh, Scotland (September 2002), pp. 61–65.
7. A. M. Davis, *Software Requirements: Analysis and Specification*, Prentice-Hall, Upper Saddle River, NJ (1990).
8. M. West, “Quality Function Deployment in Software Development,” *IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design* **180**, 5/5–5/7 (December 2, 1991).
9. J. Jackson, “A Keyphrase Based Traceability Scheme,” *IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design* **180**, 2/1–2/4 (December 2, 1991).
10. I. Alexander, “Toward Automatic Traceability in Industrial Practice,” *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering,* Edinburgh, Scotland (September 2002), pp. 26–31.
11. F. A. C. Pinheiro and J. A. Goguen, “An Object-Oriented Tool for Tracing Requirements,” *IEEE Software* **13**, No. 2, 52–64 (1996).
12. J. Cooke and R. Stone, “A Formal Development Framework and Its Use to Manage Software Production,” *IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design* **180**, 10/1 (December 2, 1991).
13. E. Tryggeseth and O. Nytrø, “Dynamic Traceability Links Supported by a System Architecture Description,” *Proceedings of the IEEE International Conference on Software Maintenance,* Bari, Italy (1997), pp. 180–187.
14. Rational RequisitePro, IBM Corporation, http://www-306.ibm.com/software/awdtools/reqpro/.
15. DOORS, Telelogic, http://www.telelogic.com/products/doorsers/index.cfm.
16. B. Ramesh and M. Jarke, “Toward Reference Models for Requirements Traceability,” *IEEE Transactions on Software Engineering* **27**, No. 1, 58–93 (January 2001).
17. J. Dick, “Rich Traceability,” *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering*, Edinburgh, Scotland (September 2002), http://www.soi.city.ac.uk/~zisman/WSTPapers/Dick.pdf.
18. P. Letelier, “A Framework for Requirements Traceability in UML-Based Projects,” *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering,* Edinburgh, Scotland (September 2002), pp. 30–41.
19. VeroTrace, Verocel Inc., http://www.verocel.com/verotrace.htm.

20. European Computer Manufacturers Association (ECMA), *Reference Model for Frameworks of Software Engineering Environments*, ECMA TR/55, Third Edition, (June 1993), http://www.ecma-international.org/publications/files/ECMA-TR/TR-055.pdf.
21. P. A. Wilcox, M. J. Smith, A. D. Smith, R. J. Pooley, L. M. MacKinnon, and R. G. Dewar, "OPHELIA: An Architecture to Facilitate Software Engineering in a Distributed Environment," *Proceedings of the 15th International Conference on Software and Systems Engineering and Their Applications*, Paris, France (2002), Volume 2, pp. 1/7–7/7.
22. S. A. Sherba, K. M. Anderson, and M. Faisal, "A Framework for Mapping Traceability Relationships," *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering*, Montreal, Canada (September 2003), http://www.soi.city.ac.uk/~gespan/paper5.pdf.
23. J. Huffman Hayes, A. Dekhtyar, and J. Osbourne, "Improving Requirements Tracing via Information Retrieval," *Proceedings of the 11th IEEE International Requirements Engineering Conference*, Monterey Bay, CA (2003), p. 138–150.
24. G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation," *IEEE Transactions on Software Engineering* **28**, No. 10, 970–983 (October 2002).
25. A. Marcus and J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links Using Latent Semantic Indexing," *Proceedings of the 25th IEEE/ACM International Conference on Software Engineering,* Portland, OR (May 2003), pp. 125–136.
26. A. Egyed and P. Grünbacher, "Automating Requirements Traceability: Beyond the Record & Replay Paradigm," *Proceedings of the 17th IEEE International Conference on Automated Software Engineering,* Edinburgh, Scotland (September 2002), pp. 163–171.
27. A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting Source Code Changes by Mining Change History," *IEEE Transactions on Software Engineering* **30**, No. 9, 574–586 (September 2004).
28. T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *Proceedings of the 26th International Conference on Software Engineering,* Edinburgh, Scotland (2004) pp. 563–572.
29. K. Czarnecki and S. Helsen, "Taxonomy and Categorization of Model Transformation Approaches," *IBM Systems Journal* **45**, No. 3, 621–646 (2006, this issue).
30. J. Huang-Cleland, C. K. Chang, and M. Christensen, "Event-Based Traceability for Managing Evolutionary Change," *IEEE Transactions in Software Engineering* **29**, No. 9, 796–810 (September 2003).
31. T. Olsson and J. Grundy, "Supporting Traceability and Inconsistency Management Between Software Artifacts," *Proceedings of the IASTED International Conference on Software Engineering and Applications*, Boston, MA (November 2002), http://serg.telecom.lth.se/research/publications/docs/99_sea_information_tool.pdf.
32. R. Freude and A. Königs, "Tool Integration with Consistency Relations and Their Visualization," *ESEC/FSE Workshop on Tool Integration in System Development,* Helsinki, Finland (September 2003), pp. 6–10.
33. R. Watson, *Smarter Requirements Management with Intelligent Traceability*, White Paper, Telelogic North America Inc., Irvine, CA (July 2003).
34. P. Kruchten, *The Rational Unified Process—An Introduction*, Second Edition, Addison-Wesley-Longman, Reading, MA (2000).
35. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use-Case Driven Approach*, Addison-Wesley, Boston, MA (1992).
36. M. Cantor, "Rational Unified Process for Systems Engineering," *The Rational Edge* (August 2003), http://www-128.ibm.com/developerworks/rational/library/content/RationalEdge/aug03/f_rupse_mc.pdf.
37. R. Paige and Y. Shaham-Gafni, *Model Composition: Development of Consistency Rules*, Modelware Report D1.5 (September 2005), http://www.modelware-ist.org/index.php?option=com_remository&Itemid=79&func=fileinfo&id=11.
38. J. Grundy, J. Hosking, and W. B. Mugridge, "Inconsistency Management for Multiple-View Software Development Environments," *IEEE Transactions on Software Engineering* **24**, No. 11, 960–981 (November 1998).
39. P. Desfray, *MDA—When a Major Software Industry Trend Meets Our Toolset, Implemented Since 1994*, White Paper, Softeam (2001), http://www.omg.org/mda/mda_files/MDA-Softeam-WhitePaper.pdf.
40. N. Aizenbud-Reshef, R. F. Paige, J. Rubin, Y. Shaham-Gafni, and D. S. Kolovos, "Operational Semantics for Traceability," *ECMDA Traceability Workshop,* Nuremberg, Germany (November 2005), pp. 7–14.
41. *MDA Guide Version 1.0.1*, OMG Document omg/2003-06-01, Object Management Group, Inc. (June 2003).
42. *MOF 2.0 Query/Views/Transformations RFP*, OMG Document ad/2002-04-10, Object Management Group, Inc. (revised on April 24, 2002).
43. *MOF Model to Text Transformation Language RFP*, OMG Document ad/2004-04-07, Object Management Group, Inc. (revised on May 27, 2004).
44. J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt, "Model Transformations in Practice," *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica (October 2005), http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf.
45. F. Jouault, "Loosely Coupled Traceability for ATL," *Proceedings of the European Conference on Model Driven Architecture Workshop on Traceability*, Nuremberg, Germany (November 2005), http://www.sciences.univ-nantes.fr/lina/atl/www/papers/ECMDATraceability05.pdf.
46. *Revised Submission for MOF 2.0 Query/View/Transformations RFP (ad/2002-04-10), QVT-Merge Group, Version 2.1*, OMG Document ad/05-07-01, Object Management Group, Inc. (August 2005).

Accepted for publication December 2, 2005.
Published online July 11, 2006.

Netta Aizenbud-Reshef
IBM Haifa Research Lab, Haifa University Campus, Mount Carmel, Haifa 31905, Israel (neta@il.ibm.com). Ms. Aizenbud-Reshef works in the Software Asset Management Group on a research project in application analysis and understanding, with a focus on service-oriented architecture transformation. She has B.A. and M.S. degrees in computer science from the Technion–Israel Institute of Technology.

Brian T. Nolan
IBM Software Group, 83 Hartwell Avenue, Lexington, Massachusetts 02421 (bnolan@us.ibm.com). Dr. Nolan is a course developer at Rational University, specializing in model-driven development. Prior to his current position, he was the regional practice lead for the Rational Unified Process for Systems Engineering. Dr. Nolan holds a Ph.D. degree in the classics from Ohio State University.

Julia Rubin
IBM Haifa Research Lab, Haifa University Campus, Mount Carmel, Haifa 31905, Israel (mjulia@il.ibm.com). Ms. Rubin works in the Model Driven Engineering Technologies Group on a research project in the area of model understanding and analysis. She has an M.S. degree in computer science from the Technion–Israel Institute of Technology.

Yael Shaham-Gafni
IBM Haifa Research Lab, Haifa University Campus, Mount Carmel, Haifa 31905, Israel (yaelsg@il.ibm.com). Ms. Shaham-Gafni works in the Model Driven Engineering Technologies Group, leading in the area of integrated solution engineering. She has an M.S. degree in computer science from the Technion–Israel Institute of Technology. ■