This article appeared in the Handbook of Software Engineering and Knowledge Engineering,Vol. III: Recent Advancements, (ed) Chang S. K., World Scientific Publishing Co., ISBN 981-256-273-7 , 2005

SOFTWARE TRACEABILITY: A ROADMAP

GEORGE SPANOUDAKIS and ANDREA ZISMAN
Software Engineering Group,
Department of Computing, City University
Northampton Square, EC1V 0HB, UK
E-mail: {gespan | a.zisman} @ soi.city.ac.uk

Traceability of software artefacts has been recognised as an important factor for supporting various activities in the software system development process. In general, the objective of traceability is to improve the quality of software systems. More specifically, traceability information can be used to support the analysis of implications and integration of changes that occur software systems; the maintenance and evolution of software systems; the reuse of software system components by identifying and comparing requirements of new and existing systems; the testing of software system components; and system inspection, by indicating alternatives and compromises made during development. Traceability enables system acceptance by allowing users to better understand the system and contributes to clear and consistent system documentation.

Over the last few years, the software and system engineering communities have developed a large number of approaches and techniques to address various aspects of traceability. Research into software traceability has been mainly concerned with the study and definition of different types of traceability relations; support for the generation of traceability relations; development of architectures, tools, and environments for the representation and maintenance of traceability relations; and empirical investigations into organisational practices regarding the establishment and deployment of traceability relations in the software development life cycle. However, despite its importance and the work resulted from numerous years of research, empirical studies of traceability needs and practices in industrial organisations have indicated that traceability support is not always satisfactory. As a result, traceability is rarely established in existing industrial settings.

In this article, we present a roadmap of research and practices related to software traceability and identify issues that are still open for further research. Our roadmap is organised according to the main topics that have been the focus of software traceability research.

Keywords: Software traceability, traceability relations, representation and

maintenance of traceability, deployment of traceability, software development process

1. Introduction

Software traceability – that 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 – has been recognised as a significant factor for any phase of a software system development and maintenance process [46], and contributes to the quality of the final product.

Typically, traceability relations denote *overlap*, *satisfiability*, *dependency*, *evolution*, *generalisation/refinement*, *conflict* and *rationalisation* associations between various software artefacts [51] (e.g. requirement specifications, software analysis, design, test models, source code), or *contribution relations* between software artefacts (typically requirement specifications) and the stakeholders that have contributed to their construction. Depending on whether traceability relations associate elements of the same artefact or elements of different artefacts, they can be distinguished into *vertical* and *horizontal* relations, respectively [36]. Another distinction is concerned with the notion of *pre-traceability* and *post-traceability* relations [26]. The former category includes relations between requirement specifications and the sources that have given rise to these specifications, i.e. the stakeholders that have expressed the views and needs which are reflected in them. The latter category includes relations between requirement specifications and artefacts that are created in subsequent stages of the software development life cycle.

Depending on their semantics, traceability relations present information that can be used in different ways in the software development life cycle. For instance, traceability relations may be used to support the assessment of the implications of changes in a system and the effective execution and integration of such changes during the development, maintenance, and evolution of a system. They may also be used to support various types of analysis that can establish whether a system meets its requirements (coverage and verification analysis), whether the requirements set for a system are those intended for it (validation). Furthermore, they may facilitate: (a) system testing by relating requirements with test models and indicating routes for demonstrating product compliance; (b) system inspection by helping inspection teams to identify alternatives and compromises made during development; and (c) system acceptance by allowing users to understand and trust specific choices that have been made about the design and implementation of a system. Finally, they may lead to the reuse of system components when these components are related to requirements of existing systems that are similar to requirements of new systems [13].

Overall, as suggested by Lindval and Sandhal [36], the establishment of traceability relations makes the documentation of a system clear and consistent, and makes the process of maintaining the system less dependent on individual experts.

Many approaches have been proposed to support software traceability. Research into software traceability has been particularly concerned with:

(a) the study and definition of different types of traceability relations [16, 19, 23, 27, 31, 35, 36, 47, 50, 61];
(b) the provision of support for their generation [1, 4, 23, 44, 39, 40, 61];
(c) the development of architectures, tools and environments for the representation and maintenance of traceability relations [10, 11, 55, 44], and
(d) empirical investigations of organisational practices regarding the establishment and deployment of traceability relations during the software development life cycle [5, 8, 26, 36, 49, 51, 62].

However, despite the wide recognition of its importance and numerous years of research, effective traceability is still rarely established in contemporary industrial settings [5, 51]. This phenomenon may be attributed to the difficulty in automating the generation of traceability relations with clear and precise semantics that could, adequately and cost-effectively, support the types of analysis necessary to deliver the benefits of traceability outlined above. Typically, most of the existing approaches, environments and tools assume either that traceability relations should be identified manually (e.g. [10, 27, 47]), or offer traceability generation techniques which cannot identify relations with a rich semantic meaning (e.g. [1, 4, 40]). In the former case, the cost of identifying traceability relations manually clearly outweighs the expected benefits of traceability and makes organisations reluctant to enforce them, unless there is a regulatory reason for doing so. In the latter case, the lack of a clear and precise semantics make the asserted relations of little use and do not provide the benefits of using traceability as described above. Therefore, the relevant techniques are not widely adopted in industrial settings.

In this paper, we present a roadmap into the state of the art and practice in requirements traceability, discuss the main scientific and technological advances in this area, present the possible ways of establishing traceability that are available by current technology, and identify issues which require further research in this field. In the course of producing this roadmap, we have tried to be as objective and inclusive as possible. However, we may have not been entirely successful, as there is always a potential for missing out existing work and presenting approaches and techniques under the inevitable influence of personal perspectives and perceptions. To this end, our roadmap should be read in a critical way.

The rest of this article is organised as follows. In Section 2, we discuss the main types of traceability relations that have been proposed in the literature and suggest a classification for these types. In Section 3, we present the main approaches and techniques for generating traceability relations from manual, semi-automatic, and

automatic perspectives. In Section 4, we outline the different approaches regarding the representation and maintenance of traceability relations in software development tools, and discuss the merit of each of these approaches. In Section 5, we discuss the various ways of using traceability relations in software development and maintenance settings, and present the implications that these ways have for other aspects of traceability, including the semantics of the deployed relations and requirements for their generation and maintenance. In Section 6, we present issues related to the semantics, establishment, representation and deployment of traceability relations which, in our view, are open to further research. Finally, in Section 7 we give a summary of the main findings of our roadmap.

2. Types of Traceability Relations

Stakeholders with different perspectives, goals and interests who are involved in software development may contribute to the capture and use of traceability information. Depending on their perceptions and needs, they may influence the selection of different types of traceability relations which are used in software development projects, and can establish project specific conventions for interpreting the meaning of such relations. As it has been suggested in [36] and [51], existing approaches and tools for traceability support the representation of different types of relations between system artefacts but the interpretation of the semantics associated with a traceability relation depends on the stakeholders. Moreover, different stakeholders are interested in different types of relations. For example, end users may be interested in relations between requirements and design objects as a way of identifying design components generated by or satisfying requirements; designers may be interested in the same type of relations but as a way of identifying the constraints represented as requirements associated with a certain design object.

These phenomena are acknowledged by Dick [19] who has also stated that typically in industrial settings the semantics of traceability relations is very shallow and it is necessary to represent *deeper* and *richer* semantic traceability relations. Pinheiro and Goguen [44] have also asserted that traceability relations should have precise semantic definition to avoid the problem of culture-based interpretations. On the other hand, Bayer and Widen [6] suggested that in order to increase the use of traceability and, therefore, compensate for its cost, traceability relations should have a rich semantic meaning instead of being simple bi-directional referential relations.

In order to overcome the lack of standard semantic interpretations of traceability relations and establish meaningful forms of semantics for traceability relations, various researchers have proposed approaches, reference models, frameworks, and classifications incorporating different types of traceability relations [2, 19, 26, 35, 36, 40, 46, 51, 68, 78]. These classifications are based on different aspects of traceability. For instance, some classifications are based on the types of the related artefacts [22,

36, 46, 78], others are based on the use of traceability information in supporting different requirements management activities such as understanding, capture, tracking, evolution, and verification [19, 27, 51], or on impact analysis [68].

In general, traceability relations can be classified as *horizontal traceability* or *vertical traceability* relations [36][1]. The former type includes relations between different models, and the latter type includes relations between elements of the same model. Another classification focusing on requirements (*requirements traceability*) has been proposed in [46]. This classification includes 18 different types of traceability relations organised in five different groups. These groups are: (a) *condition link group*, which includes relations between requirements and restrictions associated with them; (b) *content link group*, which includes relations that signify comparisons, contradictions, and conflicts between requirements; (c) *documentation link group*, which includes relations associating different types of software documents to a requirement; (d) *evolutionary link group*, which includes replacement relations between requirements (e.g. a requirement X has replaced a requirement Y in requirements document); and (e) *abstraction link group*, which includes relations representing abstractions like generalisation and refinement between requirements.

In this paper, we organise the various types of traceability relations proposed in the literature into eight main groups namely: *dependency*, *generalisation*/*refinement*, *evolution*, *satisfaction*, *overlap*, *conflicting*, *rationalisation*, and *contribution* relations. These groups are described below. In our discussion about these groups, we use the term *element* in a general way to represent the different parts, entities, and objects in software artefacts that are traceable. Examples of these elements are stakeholders, requirements statements, design components (e.g. classes, states), code statements, test data, etc. It is worth noting that this classification is not orthogonal. Thus, two elements e_1 and e_2, in different or in the same software artefact can be related by more than one type of relations.

(a) *Dependency relations* – In this type of relations, an element e1 *depends on* an element e2, if the existence of e1 *relies on* the existence of e2, or if changes in e2 have to be reflected in e1. In [51], the authors proposed the use of dependency relations between different requirements, and between requirements and design elements. In their framework, dependency relations can be used to support requirements management, express dependency between system components for low-end users, and track compositions and hierarchies of elements. An application of this approach that supports the specification and evolution of workflow management systems by using dependencies between business process objects, decision objects, and workflow system objects has been proposed in [74]. Dependency relations are

[1] In [9] the term "horizontal traceability" is defined as relations between models developed in one stage of the system development life cycle (the same model type), while the term "vertical traceability" is defined as relations between different models. However, in this paper we adopt the definition given in [36]

also suggested in [41] to support the management of variability in product and service families. In [68, 69], von Knethen et al. suggested the use of dependency relations between documentation entities (e.g. textual requirements, use cases) and logical entities (e.g. function, tasks) to assist with fine-grained impact analysis. Other forms of dependency relations are found in Spanoudakis and Zisman et al. [61, 78]. In this approach, dependency relations are called *requires-feature-in* relations and associate parts of use case specifications and customer requirements specifications. The *requires-feature-in* relations denote that a certain part of a use case cannot be realised without the existence of structural and functional features required by the requirement, or that one requirement depends on the existence of a feature required by another requirement. In Maletic et al. [39], dependency relations are called *causal conformance* and are used between software documents to represent an implied ordering in the production of the related documents (e.g., bug reports cannot be produced before implementation report). In [27], dependency relations are called *developmental relations* and are used to describe the logical structure of development and provide tracing requirements through the artefacts generated during the other phases of the software development life cycle. In the *Software Information Base (SIB),* that is an approach for building software repositories to support software reuse [14], dependency relations are realised as *correspondence* relations between requirements, design, and code artefacts. Dependency relations have also been used for requirements [2, 47], scenarios, code, and model elements [22], and to support the design and implementation of product lines [53].

(b) *Generalisation/Refinement relations* – This type of relations is used to identify how complex elements of a system can be broken down into components, how elements of a system can be combined to form other elements, and how an element can be refined by another element. In [46], these relations are classified as *abstraction links* and represent abstractions between trace requirements. In the case study developed in [51], generalisation abstractions are seen as a special type of dependency relations. In [68, 69], *Generalisation/Refinement* relations are used to represent logical entities at different levels of abstraction. Generalisation/refinement relations are also used in the approach proposed in [74] to support associations between business process, decision, and workflow system objects. In [27], they are called *containment relations* and associate requirement artefacts that together form a composite artefact. Other approaches that use generalisation/refinement relations are [35, 41, 44, 50].

(c) *Evolution relations* – Relations of this type signify the evolution of elements of software artefacts. In this case, an element e1 *evolves_to* an element e2, if e1 has been replaced by e2 during the development, maintenance, or evolution of the system. In [46], the authors suggested that this type of relations should be used to associate requirements and use specialisations of this type called *replace*, *based_on*, *formalize*, and *elaborate* relations. In 51], evolution relations specify process-related links that

are used by high-end users to document the input-output relationship of actions leading from existing elements to new (or modified) elements and, therefore, identify the origins of the elements. In TOOR [44], evolution relations are called *replace* and *abandon* relations. A *replace* relation is used to signify that a requirement has changed. An *abandon* relation is used to signify that a requirement is unnecessary (i.e., it has been discarded). In [39], evolution relations are called *non-causal conformance* relations and are used to represent the fact that different documents, or their parts, conform to each other without necessarily having clear causality between them. An example of this case is related to the existence of multiple versions of the same documents. In Gotel and Finkelstein [27], evolution relations are called *temporal relations* and are used to associate requirement artefacts in order to represent the history of their development. In the Remap project [50], evolution relations between requirements are captured by using trace rules and are represented in a knowledge base. Evolution relations are also present in SIB [14] to signify *derivations* between requirements, design, and code artefacts.

(d) *Satisfiability relations* – In this type of relations, an element e1 *satisfies* an element e2, if e1 meets the expectation, needs, and desires of e2; or if e1 complies with a condition represented by e2. In [46], this type of relations is classified within the *condition link group*, which associates restrictions to requirements and contains *constraints* and *pre-condition* links. In [51], satisfiability relations have been proposed as associations between requirements and system components (e.g. design components) and are used to ensure that requirements are satisfied by a system. The satisfaction relations are product-related links, i.e. they describe properties of elements independent of how they were created. They are used to represent that requirements are satisfied by the system and to relate one or more requirements, design, implementation elements, and compliance verification procedures. In [2], satisfiability relations are specified between use cases. In [19], in order to allow rich traceability, two types of satisfiability relations have been proposed: (i) *establishes relation*, a one-to-one relationship which represents links between main arguments of a system and the requirements satisfying these arguments, and (ii) *contributes relation*, a many-to-many relationship which represents links between arguments and requirements that contribute to the satisfaction of the arguments. In TOOR [44], requirements satisfiability is defined based on the notion of *derivation* and *refinement*: (a) if a requirement r1 is satisfied, its derived requirement r2 should also be satisfied; however, if a derived requirement r2 is satisfied, this does not mean that r1 is also satisfied; (b) if a requirement r1 refines a requirement r2, then satisfying r2 implies on satisfying r1. CORE [17] tool also supports satisfiability relations between design and requirements artefacts.

(e) *Overlap relations* – In this type of relations, an element e1 *overlaps* with an element e2, if e1 and e2 refer to common features of a system or its domain. In [61, 78], overlap relations are used between requirement statements, use cases, and

analysis object models, while in 16 such relations are used between goal specifications represented in *i** models, use cases, and class diagrams. In the classification given in [46], an overlap relation is a documentation link that relates requirements with different types of documents such as test case, purpose, comment, background information, and examples. In [68, 69], overlap relations are called *representation* relations and represent relationships between document entities representing the same logical entity. The approach proposed in [22, 23] uses overlap relations between scenarios specifications and other model elements such as data flow, class, and use case diagrams. In PuLSE-BC (*Product Line Software Engineering Baselining and Customisation* [6, 7]), potential traceability relations are defined based on overlaps of models that are identified based on name matching. The overlap relations between source code and requirements or manual documents proposed in [4] and [40] are identified using probabilistic and vector space IR techniques. In [27] overlap relations are called *adopts relations* (a subtype of *connectivity* relation) and are used to associate requirement artefacts.

(f) Conflict relations – This type of relations signifies conflicts between two elements e1 and e2 (e.g., when two requirements conflict with each other). Conflict relations have been proposed in [2, 46, 51, 73, 79]. In [51], conflict relations are used to signify conflicts between requirements, components, and design elements, to define issues related to these elements, and to provide information that can help in resolving the conflicts and issues. This information is recorded by using specialised conflict resolution relations, namely *based_on*, *affect*, *resolve*, and *generate* relations between requirements artefacts and the rationale, decisions, alternatives, and assumptions associated with them. In [46], conflict relations are classified in the *content link group*. In [32, 79], conflicts are represented by *inconsistency* relations between requirements and design artefacts and are identified by using a goal-based approach. In this approach, inconsistency relations are established when similar goals cannot be achieved in two different specifications, or in parts of the same specification. Inconsistency relations between design artefacts are also used in [76].

(g) *Rationalisation relations* – Relations of this type are used to represent and maintain the rationale behind the creation and evolution of elements, and decisions about the system at different levels of detail. In [51], rationalisation relations are captured based on the history of actions of how elements are created. In [35], rationalisation relations are expressed between traceable specifications (a software specification with different level of granularity such as document, model, diagram, use case, etc.) and a rationale specification (a document containing assumptions or alternatives to a traceable specification). Remap [50] also captures design rationale and represent them in a knowledge base. Rationalisation relations are also found in [46] and [54].

(h) *Contribution relations* – Relations of this type are used to represent associations between requirement artefacts and stakeholders that have *contributed* to

the generation of the requirements. Contribution relations were initially proposed by Gotel and Finkelstein [27] to support requirements pre-traceability. Pre-traceability is the ability to relate a requirement (called "contribution") with the stakeholders that expressed it and/or contributed to its specification (called "contributors"). Their approach identifies three different types of contributors: (a) *principals* who motivate the production of artefacts are committed to what is expressed in the artefacts, and are responsible for the consequences of the artefacts for the system; (b) *authors*, who choose, formulate, and organise the content and structure of the information in the artefacts; and (c) *documentors*, who capture, record, or transcribe the information in the artefact.

Table 1 presents a summary of the various approaches for the different types of traceability relations that have been proposed in the literature and the types of the software artefacts that these types may interrelate. We also represent associations between the different types of artefacts and stakeholders. In this table, software artefacts, or their parts, are distinguished depending on the phase of the software development life cycle that they are created in. More specifically, we classify artefacts as: (a) requirements, (b) design, (c) code, and (d) others (e.g. goal documentation, test cases, rationale and purpose documentation, etc). The columns of the table represent the eight different types of traceability relations discussed above, the rows represent combinations of different types of software artefacts, and the cells indicate the approaches that realise (in some form) the specific type of relation between the relevant types of artefacts. Empty cells in the table signify combinations of traceability relation and artefact types which, to the best of our knowledge, are not realised by any of the approaches that have been proposed in the literature.

From Table 1, it is clear that most of the existing approaches have proposed different types of traceability relations that relate requirements specifications [2, 6, 14, 19, 22, 27, 35, 41, 44, 47, 50, 51, 69, 78], and requirements with design specifications [14, 16, 17, 22, 35, 50, 51]. Some approaches have proposed traceability relations between code specifications and requirements and design artefacts [4, 22, 39, 51]. We attribute this to the fact that traceability was initially proposed to describe and follow the life of a requirement (i.e. *requirements traceability* [27]). In addition, the establishment of traceability relations involving code specifications and other software artefacts is not an easy task. It should also be noted that very few approaches support conflict [2, 32, 47, 51, 73, 79] and rationalisation relations [35, 50] despite the importance of these types of relations. In addition, only one approach has focused on contribution relations between stakeholders and requirements [27] despite the fact that contribution relations are important to establish the source of the requirement and to identify stakeholders that should be consulted in the case of changes to these requirements.

The different types of traceability relations that have been proposed in the literature and the lack of a commonly agreed standard semantics for all these types do

not provide confidence in the use of traceability techniques and do not facilitate the establishment of a common framework to allow the development of tools and techniques to support automatic (or semi-automatic) generation of these relations. In our view, the establishment of standardised definitions for different types of traceability relations are necessary to: (a) assess the accuracy of established relations of these types in specific projects, and (b) develop tools and techniques to support the generation, maintenance and deployment of such relations.

Table 1: Summary of the different types of traceability relations for different artefacts generated during various phases of the software development life cycle

Rel. Type ***Artefacts***	*Dependency*	*Generalisation* *Refinement*	*Evolution*	*Satisfiability*	*Overlap*	*Conflict*	*Rationalisation*	*Contribution*
Stakeholders - Requirements								Gotel et al [27]
Stakeholders - Design								
Stakeholders - Code								
Stakeholders - Other								
Requirements - Requirements	Alexander [2] Gotel et al [27] Pro-Art [47] Ramesh et al [51] Von Knethen et al [69] Rule-based tracer [78]	Gotel et al [27] Letelier [35] Mohan et al [41] TOOR [44] Remap [50] Pro-Art [47] Von Knethen et al [69]	SIB [14] Gotel et al [27] TOOR [44] Remap [50] Pro-Art [47]	Alexander [2] Dick [19] TOOR [44] Pro-Art [47]	Bayer et al [6] Egyed [22] Gotel et al [27] Rule-based tracer [61, 78]	Alexander [2] Pro-Art [47] Ramesh et al [51]	Letelier [35] Remap [50]	
Requirements- Design	SIB [14] Egyed [22] Ramesh et al [51]	Letelier [35] Remap [50]	Remap [50]	Ramesh et al [51] CORE [17]	Cysneiros et al [16] Egyed [22]	Kozle-nkov & Zisman [32] Ramesh et al [51]	Letelier [35] Remap [50]	
Requirements- Code	Egyed [22] Maletic		Maletic et al [39]	Ramesh et al [51]	Antoniol [4]			

	et al [39]							
Requirements-Other		Letelier [35] Mohan et al [41] TOOR [44]		Dick [19] Ramesh et al [51] TOOR [44]	Cysneiros et al [16] Pro-Art [47]	Ramesh et al [51]	Letelier [35]	
Design - Design	Egyed [22], Von Knethen et al [69]	Von Knethen et al [69]		Ramesh et al [51]		Xlinkit [73] Zisman et al [79] Ramesh et al [51]		
Design - Code	SIB [14] Egyed [22], Maletic et al [39]	Letelier [35]	SIB [14] Maletic et al [39]	Ramesh et al [51]			Letelier [35]	
Design - Other		Letelier [35]		Ramesh et al [51]		Ramesh et al [51]	Letelier [35]	
Code - Code				Ramesh et al [51]				
Code - Other	Maletic et al [39]		Maletic et al [39]	Ramesh et al [51]				
Other - Other	Xu et al [74]	Letelier [35] Xu et al [74]		Ramesh et al [51]	Rule-based tracer [78]		Letelier [35]	

As described in the beginning of this section, reference models have also been proposed to support semantic interpretation of traceability relations. Ramesh et al. [51] have proposed the use of metamodels to support the representation of traceability information including types of different traceable elements and types of relations that may exist between these elements. The metamodel that they have proposed for this purpose is composed of entities and relationships that can be specialised and instantiated to represent traceability models used in specific organisations or projects. More specifically, their metamodel contains three types of entities. These are: (i) *object entities*, that signify the conceptual elements which may be related by traceability relations (e.g. requirements, assumptions, designs, rationale, system components, etc); (ii) *stakeholder entities*, that represent the agents involved in the system development and maintenance (e.g. project managers, system analysts designers, etc); and (iii) *source entities*, that represent the documentation of traceability information (e.g. requirements specifications, meeting minutes, designed

documents, etc). The relationships in their metamodel associate object instances through *TRACES-TO* links, stakeholders and source instances through *MANAGES* links, source and objects instances through *DOCUMENTS* links, and stakeholders and objects or stakeholders and traceability links through *HAS-ROLES-IN* links. These relationships can be specialised depending on the application or the views of the stakeholders.

Applications and extensions of the traceability model and traceability links suggested in [51] have been proposed to assist in the specification and evolution of workflow management systems [68], identify common and variable requirements in product and service families [41], and assist software development process based on UML [35]. Another metamodel that supports capture of design rationale has been proposed in [50].

With regards to support for different types of traceability relations, the study reported in [51] shows that many CASE tools do not support the identification of satisfiability relations. This is due to the difficulty in automating the specifications of relations that can be identified based on existing relations (e.g. by transitivity) and deficiency in representing satisfying requirements as well as the degree of satisfaction. Dependency relations are also minimally supported by traceability tools and there is lack of precise characterisation of dependency types and the strength of the relevant relations. On the other hand, software configuration management tools provide ways of representing and enacting evolution of coarse-grained artefacts, as well as notifying the users about changes. However, traceability tools only manage fine-grained objects. The study has also demonstrated that even experienced traceability users find difficult to capture complex traceability information. The authors suggested that it is necessary to develop abstraction mechanisms to support different granularity and sophistication when performing traceability, inference services to support semantics of traceability link types and access to large amount of traceability information, formal and informal trace description, and mechanisms to define and enact model driven trace process.

3. Generation of Traceability

The majority of contemporary requirements engineering and traceability tools offer only limited support for traceability as they require users to create traceability relations manually. The manual creation of traceability is expected in numerous techniques and approaches including [1,2, 5,6, 19, 2744, 47] and tools (e.g. CORE [17], DOORS [64], PuLSE-BC [6], RDT [52], RTM [30], RETH [31]). As the manual creation of traceability relations is difficult, error-prone, time consuming and complex, despite the advantages that can be gained, effective traceability is rarely established manually unless there is a regulatory reason for doing so. To alleviate this problem, some approaches which support automatic or semi-automatic generation of

traceability relations have been proposed (see [4, 12, 22, 29, 34, 40, 44, 47, 50, 61, 78]).

In the following, we describe the different approaches to the generation of traceability relations, grouped according to the level of automation that they offer. This level ranges from manual, to semi-automatic, and fully automatic generation. We also compare these approaches with respect to the level of effort that they require in establishing the traceability relations, the complexity of their use, the maturity of their development, and the precision of the relations that they can generate.

3.1 Manual generation of traceability relations

In this group of approaches, manual declaration of traceability relations is normally supported by visualisation and display tool components, in which the documents to be traced are displayed and the users can identify the elements in the documents to be related in an easier way. Examples of this situation occur in tools like RETH [31], DOORS [64], RTM [30], RDT [52], and extensions of DOORS such as [2, 19]. The majority of these approaches claim to support (semi-)automatic traceability generation, since the relationships are identified manually, but the links between the elements being related are automatically generated by the supporting tool based on these relationships. However, in our opinion, these approaches are considered manual, as they expect the user to identify and mark the elements to be traced. In addition, some of these tools provide no support for defining the semantics of traceability relations in a declarative and enforceable axiomatic form (e.g. RETH [31], DOORS [64] and RTM [30]).

The approach in [1, 2] has been implemented on top of the requirements management tool DOORS [64] to allow generation and organisation of traces into groups such as use cases and requirements, instead of having traces belonging only to atomic objects (individual requirements). Here traceability hyperlinks are created based on the manual specification of satisfiability, dependency, and conflict relations (see Section 2) between use cases and requirements, and fuzzy matching between use case names and references. The relationships are specified by the use of a special-purpose screen in which the users relate the names and object identifiers of groups of use cases and requirements. The HTML hyperlinks are constructed by the tool and can be visualised and navigated by using a web browser. The rich traceability approach proposed in [19] has also been implemented on top of DOORS [64]. It uses a rich traceability explorer tool to support visualisation of the objects being traced and the traceability links.

In [27], the contribution relations (see Section 2) are manually defined in the artefacts, in an interactive way, and represented as hypertext mark-ups using the Standard Generalized Markup Language (SGML). A prototype tool has been developed to support visualising and querying the artefacts stored in an online

repository, agents, and relations; interactive definitions of traceability relations and agent details; and inference of agent capacities, social roles and commitments based on pre-defined rules.

The approach in [6] (PuLSE-BC) is still theoretical and tools to support automatic generation of *potential* traces between product family metamodels (e.g. views, models, and life cycle stages such as scoping, architectural, and implementation), based on name matching, have not yet been implemented.

Although the approaches in [2] and [19] provide the users with advanced support for visualising and navigating through the generated traceability relations, the effort to establish these relations is still high, especially when dealing with large and complex artefacts. The correctness of the traceability relations generated in [1] and [2] relies on understanding the semantics of the relations by the users who identify them. And as this understanding can differ between the users involved in the process, different interpretations and inconsistencies may arise when referring to the relations.

3.2 Semi-automatic generation of traceability relations

In order to overcome the issues associated with the manual generation of traceability relations, some approaches have been proposed in which traceability relations are generated in a semi-automatic way. We organise the semi-automatic traceability generation approaches into two groups: (a) *pre-defined link group*, that is concerned with the approaches in which traceability relations are generated based on some previous user-defined links [11, 12, 22, 23], and (b) *process-driven group*, that is concerned with the approaches in which traceability relations are generated as a result of the software development process [46, 47].

An example of a *pre-defined link* group approach has been proposed by Egyed and Gruenbacher [22, 23]. Egyed and Gruenbacher suggest the use of a rule-based scenario-driven approach to support the generation and validation oftrace dependencies between scenarios, code, and model elements such as data flow, class, and use case diagrams. In this approach, traceability dependencies are generated and validated based on observed scenarios of the software system being developed, and on manually defined *hypothesised traces*, that link artefacts with these scenarios. Manually detected overlaps between scenarios and model elements of the system are represented as a footprint graph, which is normalised and refined. This footprint graph is used to support automatic generation and validation of new traceability links between model elements, model elements and code, and model elements and scenarios based on rules. The new trace dependencies are derived based on: (a) *transitive reasoning* (i.e., if A depends on B and B depends on C, then A depends on C); and (b) *share used of common ground (code)* (i.e., the use of the criterion that if A and B depend on subsets of a common ground and these subsets overlap, then A depends on B). This approach has been validated on a library loan system and on a

video-on-demand system [23].

In [11, 12], Cleland-Huang et al. proposed an event-based approach to support generation of traceability links between requirements and performance models [12], and between non-functional requirements and design and code artefacts [11]. Similar to [22], in their approach fine-grained traceability links are dynamically generated during system maintenance and refinement based on user-defined links. These user-defined links are specified during inception, elaboration, and construction of the system. Their event-based technique supports dynamic trace generation based on invariant rules of design patterns which are used to identify critical components of classes.

PRO-ART [47] is an example of a process-driven approach in which traceability relations are generated as a result of creating, deleting, and modifying a product by using development tools (process execution environment). The development tools must ensure: (a) the recording of execution, input and output of each action related to the creation, deletion, and modification of a product in a trace repository, (b) generation of dependency links between two dependent objects, and (c) recording of the stakeholder performing the action and relationships between the action being executed and previous actions.

Although the above approaches may be considered an improvement when compared with the manual approaches, the identification of the initial user-defined links required by some of the approaches may still cause traceability to be error-prone, time consuming, and expensive. In addition, the tools that have been developed to support these approaches [12, 22] are prototypes implemented to illustrate and evaluate the approaches and not to be used in large-scale industrial settings. In the case of the process-driven approach, the generated traceability relations are dependent on the way that the system is developed.

3.3 Automatic generation of traceability relations

Recently, there have been proposals of approaches to support automatic generation of traceability relations. Some of these approaches use information retrieval (IR) techniques [4, 29, 40], others use traceability rules [50, 61, 78], special integrators [55], and inference axioms [45].

The use of information retrieval techniques to support generation of traceability relations has been proposed in [4, 29, 40]. In [4], a traceability relation is established between a requirement document and source code component, if the document matches a *query* specified as a list of identifiers extracted from the source code. Depending on the similarity between queries and documents, a ranked list of documents for each source code component is produced. Queries are matched with documents using *probabilistic* and *vector space* IR techniques. This approach is based on the assumption that the vocabulary of the source code identifier overlaps with

various items of the requirements documents due to the fact that programmers normally choose names for their program items from the application-domain knowledge. This work has been reported to produce traceability relations at low levels of precision and reasonable levels of recall. It should be noted, however, that the relations produced by this approach can only represent overlap relations between elements in different system artefacts that refer to common features of a system.

Another approach that automates the generation of traceability relations using vector space IR techniques has been proposed in [29]. This approach attempts to reduce the number of missed and irrelevant traceability relations by using a classical vector IR model technique, and a classical vector IR model technique extended with the use of key-phrase lists or the use of thesauruses. The study has demonstrated that the use of a key-phrase list can improve the recall of the generated relations (fewer missed relations), but decreases their precision (i.e., it generates more irrelevant relations) when compared to classical vector IR techniques. It has also demonstrated that the use of a thesaurus outperforms in terms of recall and, sometimes, also in terms of precision the use of key-phrases.

The approach proposed in [39] is based on the use of *Latent Semantic Indexing* (LSI) to generate traceability links between system documentation (e.g., manual, requirements, design or test suites) and source code. This approach does not depend on the specification language used to produce the documentation of a system and the programming language used in the source code. It takes into consideration synonym terms since it uses linear combinations of terms as dimensions of the representation space. In this approach, a corpus is built based on pre-processing of the documents and source code. A traceability link between two documents is established when the semantic similarity measure of these documents is greater than a threshold. A comparison of this work with the work in [4] demonstrates that both recall and precision results are better in [40].

In the rule-base tracer proposed in [61, 77, 78], traceability relations between requirements statements, use cases, and analysis object models are automatically generated by using XML-based traceability rules. The rules are used to identify syntactically related terms in the requirements and use case documents with semantically related terms in an object model. The documents to be traced are represented in XML and the generated relations are represented as hyperlinks and expressed as an extension of Xlink [18]. The approach has been evaluated in case studies for a family of software-intensive TV systems and for a university course management system. The levels of recall and precision achieved by this approach in the reported case studies are promising: recall and precision measures range between 50% and 95%. These results provide evidence of the ability of the approach to support automatic generation of traceability relations.

Another approach that supports the automatic generation of traceability relations between requirements and design artefacts based on *trace rules* has been proposed in

the Remap project [50]. This approach also supports arbitrary chaining of rules in which the conclusion part of a rule can become part of the condition part of another rule. The generated traceability information is maintained in a knowledge base and may be used for further reasoning. An extension of Remap that allows the identification of commonality and variability traceability management in the development of e-service families (e.g. Internet, wireless and land-based telecommunication systems) between customer requirements and design artefacts has been proposed in [41].

In [55], Sherba et al. have proposed an approach that allows the generation of new traceability relations based on relationship chaining. This approach uses special *integrators*, which can discover and create traceability relations between software artefacts and other previously defined relations. The new identified relations can be generated based on indirect and transitivity dependencies, complex dependencies containing more than one source or destination elements being related (anchors), intersection of anchors, or matching of pre-defined conditions between artefacts and/or relations. When more than one chaining option is available for certain documents, the user has to choose a specific chain of relation type. At the time of writing of this paper, this approach was still in early stage of its development and no prototype tool was implemented.

In TOOR [44], traceability relations are defined and derived in terms of axioms. Based on these axioms, the tool allows automatic identification of traceability relations between requirements, design, and code specifications. TOOR also supports derivation of additional relations from the axiomatic definitions by transitivity, reflexivity, symmetry, extraction, and dependency. In addition, it allows users to define traceability relations manually.

It has to be appreciated that, although none of the above approaches can fully automate the generation of traceability relations, they have taken significant steps towards this direction. However, the achievement of acceptable levels of recall and precision that would increase the trustworthiness of the generated links is still missing. At the moment, there is no consensus of what could be considered as satisfactory recall and precision rates in industrial settings.

Also, existing approaches do not always deliver the relations that they can generate at adequate performance levels (see Section 6). It is also important to note that the majority of the developed approaches (e.g. [4, 29, 40]) have been implemented only as prototype tools and, therefore, they have not achieved a level of maturity required for large-scale use.

The approaches described in [44] and [61] are easy to use once a complete set of traceability relation generation rules and axioms have been identified. This, however, is not always easy. As a way of addressing this problem, in [58] the authors have proposed a machine learning algorithm, which generates new traceability rules, based on examples of undetected traceability relations identified by the users.

4. Representation, Recording and Maintenance of Traceability Relations

The representation, recording and maintenance of traceability relations in different tools and environments are supported by a wide range of architectural approaches. These approaches can be distinguished into five main types, depending on the level of integration that they assume between the artefacts and traceability relations, and the representation framework that they use to store the artefacts and relations. These approaches are:

(a) the single centralised database approach;
(b) the software repository approach;
(c) the hypermedia approach;
(d) the mark-up approach; and
(e) the event-based approach

The main characteristics of each of these approaches and their merit in supporting traceability are discussed in the following. We also give examples of requirements management, CASE and traceability tools, and environments that realise each of the above approaches along with brief overviews of the ways in which they do it.

4.1 Single centralised database approach

In this approach, both the artefacts and the traceability relations that can be created between them are stored in a centralised database, which typically underpins the tool that is used to maintain the traceability relations. Most of the industrial requirement management tools that support traceability advocate this approach (e.g. DOORS [64], RTM [30]). Typically, these tools store the traceability relations between artefacts in an underlying relational database [51]. However, there are also tools based on object-oriented (e.g. TOOR [44]) or proprietary database technology.

The main benefit of using an underlying database is that the processing of the recorded relations can be based on extensive and efficient querying facilities that are available from this database. It has, however, to be appreciated that this approach makes it difficult to record and maintain traceability relations between artefacts which are not generated by the tool that manages the relations. To alleviate the problem of recording, some of the tools provide artefact importing and exporting capabilities (e.g. [30, 64]). Such facilities give some degree of flexibility, but are normally available only for artefacts created by other tools of the same vendor or by tools which operate on a common artefact representation framework (DOORS [64], for instance, provides import and export capabilities for artefacts which are constructed and managed by most of the popular CASE tools). It should be noted, however, that import and export facilities cannot support the effective maintenance of traceability

relations between evolving artefacts, which are maintained by separate tools.

Furthermore, it has to be appreciated that, depending on the type of the underlying database, this approach may make it difficult to differentiate between different types of traceability relations. In tools which use relational or proprietary databases, for instance, it is difficult to define different types of relations and specify constraints for monitoring the integrity of the generated relations.

4.2 Software repositories

A second approach is to record traceability relations in a centralised software repository along with the artefacts that they relate [14, 46]. The main difference of this approach with the single database approach is that software repositories provide sufficient flexibility for defining schemas for storing a wide range of software artefacts and traceability relations between them. In addition, software repositories provide application programming interfaces (APIs) which implement data definition, querying, and management facilities that may be used to link them in client-server architectures with other tools which are used to construct the involved software artefacts.

The Software Information Base (SIB) [14] is an experimental repository system that can support the definition of complex semantic structures for holding information about artefacts and traceability relations at an infinite number of classification layers based on the data model of the conceptual modelling language TELOS. SIB defines traceability relation types that correspond to what in this paper has been termed as *dependency* and *evolution* relations. It also allows the definition of arbitrary additional types of traceability relations, and provides an API through which it may be connected as an information server to external tools.

PRO-ART [47] is a process centred requirements engineering environment which also advocates the software repository approach. PRO-ART assumes a process centred integration of tools that are used to create and maintain the involved software artefacts. More specifically, PRO-ART assumes the explicit specification of processes, which can create the involved artefacts. These processes are defined in terms of artefact creation, deletion, and modification actions. These actions are realised by different tools operating on the top of the underlying PRO-ART repository. Traceability relations in this environment are recorded as a by-product of executing such actions on the artefacts that they are meant to relate. For example, when a developer creates a textual rationale for an object class, PRO-ART automatically creates a *rationalisation* relation between the class and the textual annotation. In PRO-ART, the text providing the rationale may be created by text editor that is different from the tool that was used to create the related class. However, by virtue of monitoring the enactment of an underlying process model, PRO-ART is able to identify the purpose of invoking the text editor and create the

relevant rationalisation relation. The main benefit of this process-centric approach is that it allows the definition of ways of capturing traceability relations as part of software development processes and, thus, it can monitor or enforce the systematic capture of such relations (this need has been suggested by case studies – see [20] for example).

Overall, the software repository approach requires a heavy up-front tool integration effort to support traceability that may not be desirable, or feasible, in distributed software development settings. Furthermore, in realisations of this approach along the lines suggested by PRO-ART, additional effort is required for modelling the processes supported by the individual tools and their co-ordination of these processes in the context of a software development life-cycle model.

4.3 The hypermedia approach

To solve the problem of maintaining traceability relations as the artefacts that they associate evolve without having to integrate the relevant tools around a software repository, some tools advocate an approach based on *open hypermedia* architectures [71].

Sherba et al [55] have developed a traceability management, called *TraceM* that realises this approach. *TraceM* is a research prototype system at an experimental development stage, which supports the recording, maintenance, and traversal of relations between software artefacts that are constructed by heterogeneous tools. *TraceM* stores traceability relations separately from the artefacts that they associate. A traceability relation in this system can be defined as an n-ary association between artefacts of different types, or their parts, by using metadata. These metadata specify: (i) the *types* of the artefacts associated by a relation, (ii) the *external tools* that create these artefacts, (iii) *transformers* that can be used to transform artefacts into the common representation framework of *TraceM*, and (iv) *integrators* which can be used to automatically discover and create the traceability relations. Metadata are also used to specify the types of stakeholders which may be interested in different types of traceability relations and the stages in a project when these relations are needed. Using a *scheduling service* that is provided by the system, developers can also specify the conditions for invoking artefact translators (e.g., a translator may be invoked any time when a new version of an artefact is created) and integrators. To take full advantage of its services (e.g. to navigate between artefacts using recorded traceability relations), developers have to integrate the external tools that are used for creating artefacts with *TraceM*. This integration is possible using standard techniques available in open hypermedia environments. In cases of artefacts which are created by non integrated external tools, developers may use *TraceM* to record and view relations.

4.4 The mark-up approach

To enable traceability in widely distributed and heterogeneous software engineering settings, some systems advocate the use of representations of traceability relations using mark-up languages and store these relations separately from the artefacts that they associate.

Gotel and Finkelstein [27], have developed a toolkit that can be used to create and maintain *contribution* relations. This toolkit uses a combination of HTML and descriptive mark up representations to store different types of contribution relations between artefacts as hyperlinks (i.e. relations with hypertext trace anchors that can provide the required navigational capabilities). Maletic et al. [39] have also developed a tool in which traceability relations are stored as "hyperlinks" of different types. These types specify the arity, directionality, and traversability of the relations.

STAR-Track [56] is a web-based requirements tracing tool that uses tagging mechanisms to represent traceability relations. The tags in *STAR-Track* can represent document elements or relations between these elements. Each tag is composed of a document identifier and a title. The tagging method used in this approach is quite simple: users can either have the different sections in a document used as tags or define their own tags. The relations are denoted by the tag identifiers of the document elements related by them.

In the rule-based tracer described in [61, 78], both traceability relations and the artefacts that they relate are represented in XML. Traceability relations, however, are recorded separately from the artefacts they relate, and *XLink* elements [18] are used to indicate the parts of the artefacts these relations refer to. This system also incorporates *translators* that can transform textual artefacts from their original format into XML, and supports standardised XML representations for other types of models (e.g. XMI for object models). The rule-based tracer does not require any form of tool integration and, at its current state of development, provides only primitive support for maintaining traceability relations when artefacts are modified. In such cases, the modified artefacts have to be re-translated into XML and the tool embarks in a full rule-based analysis of their contents to identify the traceability relations, which remain valid or emerge after the changes.

4.5 The event-based approach

Except from the centralised software repository approach, in all other approaches explained above, it is difficult to ensure that traceability relations between artefacts will always be updated after modifications of these artefacts.

As a solution to this problem, Cleland-Huang et al. [10] have developed an event-based traceability (EBT) server for recording and maintaining traceability relations between requirements documents and other software artefacts. This system

is based on an event-notification mechanism that implements the *observer* pattern [25]. More specifically, the requirement documents can register their dependencies to other artefacts using the registry of the system. Following the registration of dependencies, their system monitors the artefacts and when any of them is modified, it notifies all the dependent requirements about the change. The requirement documents have responsibility for updating their contents if necessary. This system can be used for maintaining *dependency* relations once they are identified. However, it provides no support for identifying them.

In terms of their support for interoperability, the event-based and mark-up approaches appear to be superior to centralised database and the software repository approach. The latter, however, appear to be stronger in terms of performance and offer better data management facilities.

5. Deployment of Traceability

Traceability relations may be deployed in the development life cycle of a software system to support different development and maintenance activities, including:

- change impact analysis and management [12];
- system verification, validation, testing and standards compliance analysis [51];
- the reuse of software artefacts [69]; and
- software artefacts understanding [4, 40, 50, 51].

In the following, we overview ways in which traceability may support the above forms of analysis and discuss the main types of relations that can be used in these forms. We also discuss factors than can promote or prohibit the use of traceability in industrial settings as reported by relevant case studies [5, 49, 51].

5.1 Traceability for change impact analysis and change management

One of the primary drivers for establishing and maintaining traceability relations between different artefacts developed to document and implement software systems is the ability to use these relations during the entire life-cycle of a system in order to: (a) establish the impact that potential changes in some part of the system may have in other parts (i.e., *change impact analysis*), and (b) make decisions about whether or not such changes should be introduced, and with what priority (i.e., *change management*).

The simplest form of analysing the impact of a change in a given artefact (e.g. a requirement statement) is the identification of all the other artefacts that will be affected by the change (e.g. design artefacts and code). Primitive change impact analysis requires the provision of basic querying facilities to retrieve traceability relations of specific types that may also have specific values for the properties defined for these types. Most of the existing traceability tools and environments

provide such querying facilities.

However, more complex forms of change impact analysis may also be desired in different settings. Examples of these forms are: (a) the classification of affected artefacts into different groups subject to the exact effect that the change will have on them, (b) the identification of side-effects that the change may have, and (c) the estimation of the cost of propagating the change. The delivery of such capabilities requires support for the composition of different traceability relations into *trace-paths*. These trace-paths can demonstrate how impact is propagated across artefacts that are not directly related. Compositions of traceability relations may be established by evaluating *regular expressions* [44], *deductive rules* [51], or *traceability rules* [61]. Some research prototypes provide such composition capabilities (e.g. TOOR [44], PRO-ART [47], and the rule-based tracer [61]). Most of the industrial traceability tools, however, can realise these capabilities only through the generation of appropriate scripts that can compute the required compositions. Support for the estimation of the cost of executing and integrating requested changes is provided at an inferior level. This is because the few cost estimation models (see [33] for example) that have been developed for this purpose cannot provide very precise cost predictions.

In certain cases, the assessment of the impact of a change may also require the execution of a simulation model to demonstrate the potential effects of the change. This kind of analysis requires not only the establishment of specific types of traceability relations between the artefacts to be changed and the simulation model that can demonstrate the effects of the change, but also the ability to propagate changed values across these relations, to inject changed values into the simulation model, and to execute the change. The EBT system [10] offers such capabilities and has been used to analyse the effect of requirement changes onto the performance of software systems by using dependency relations between requirements and performance simulation models [12].

Clearly, the accuracy of both simple and complex forms of impact analysis depends on the semantics, granularity and accuracy of the traceability relations, which are taken into account. Relations that are established by virtue of identifying references to common entities in different software artefacts may provide some evidence that a change could potentially have an impact on an artefact, but cannot establish this impact, or its nature, with certainty. Examples are the *overlap* relations between requirements and object-oriented analysis models identified in [61, 78], or between source code artefacts and manual pages or requirements identified by information retrieval techniques in [4, 40]. On the other hand, traceability relationships with a rich semantic content (e.g. the *dependency* relations in [51] or the *requires_execution_of* relations in [61]) can lead to more accurate impact analysis results.

The necessity of having traceability relations with rich semantics in order to be

able to perform accurate impact analysis has been clearly identified by empirical studies [8, 36]. Empirical research has also indicated that the accuracy of impact predictions depends on the granularity of the entities, which are associated by traceability relations. As reported in [8], fine-grain relations that associate specific entities/parts within broad models and documentation result in more accurate results. Industrial case studies have also indicated that software developers are often reluctant to rely on traceability relations that have been produced automatically due to doubts about the correctness of such relations [36].

5.2 Traceability for software validation, verification, testing and standards compliance

Depending on their semantics, traceability relations can provide the basis for performing different types of analysis in order to ensure that a system implements the requirements desired by the stakeholders involved in its development (*validation*), verify that it satisfies certain properties and its specification (*verification*), test its individual components and the system as a whole, and assess its compliance with respect to existing standards.

Pre-traceability *contribution* relations, for instance, may be used to identify stakeholders and involve them in requirement validation activities [27]. The ability to verify the satisfaction of the specification of a system and its required properties also depends on the existence of traceability relations with appropriate semantics. Preliminary system verification, for instance, can be performed by consulting and/or composing *refinement*, *dependency* and *satisfiability* relations to establish whether all the requirements of a system have been allocated to specific design and/or source code components [42] and [69]). Similarly, traceability relations may be used to check the existence of appropriate test cases for verifying different requirements and to retrieve such cases for further testing. The results of this preliminary verification analysis typically provide input to software inspection and auditing procedures [69]. They can also be correlated with the results of other forms of analysis, which may be carried out as part of inspection and auditing procedures (e.g. functional simulations, textual and syntactic analysis of code, and standards auditing [42]). Traceability relations may also be used in system reviews concerned with the assessment of requirements and design models. Haumer et al. [28], for instance, use *goal attainment* and *failure* pre-traceability relations between goal oriented requirement models and collections of observed cases of system usage encoded in multimedia (e.g., video and audio), in order to inform review activities which are concerned with the assessment of adequacy of these models.

Traceability relations have also been used in automated forms of verification analysis, which are aimed at assessing the consistency of different models of the same system [21]. Spanoudakis et al. [59], for instance, present an approach that can be

used to rewrite formal requirement specifications in order to express *overlap* relations known to hold between their parts and make them amenable to consistency checking using theorem proving. Techniques for detecting overlap relations between structural and behavioural object-oriented models of software systems, and deploying them for checking the consistency of these models with respect to specific consistency rules are also discussed in [60]. In [24], Fiutem and Antoniol extract design models from the source code of a system, detect *overlap* relations between these models and the original design models that had been developed prior to implementation by using string matching algorithms, and use these relations to verify the consistency of the implementation with the original design.

5.3 Traceability for software reuse

Researchers have widely acknowledged the potential of traceability relations in identifying reusable artefacts in the software development life-cycle [4, 14, 69]. These artefacts may be at different levels of abstraction (e.g. source code, design or requirement artefacts) and can be identified and reused through different scenarios, which require the existence of different types of traceability relations.

In the context of the *Software Information Base* [14] *dependency* traceability relations, called "correspondence" relations, are used to associate requirements specifications with design models, and design models with source code. These relations are established in the context of *application frames* which group artefacts of specific (or families of) software applications and must be asserted manually. Software engineers can follow these relations in order to locate concrete reusable artefacts in an application frame at low levels of abstraction (i.e., design and source code artefacts) once the possibility of reusing this frame (or parts of it) has been established. This possibility is established by assessing the similarity of application frame requirements with requirements for new systems. Similar approaches have been suggested in [19] and [42]. In [42], reusable design elements are identified through *satisfiability*, *refinement* or *overlap* relations that connect them with events, pre-conditions, and post-conditions in use case models. Also, as suggested in [4], traceability relations between source code artefacts (e.g. functions in code libraries) and manual pages can help software engineers understand the functionality of the former and appreciate the possibilities of using (or re-using) them in specific contexts.

The traceability relations used in all the above approaches are *vertical relations* 36. Different opportunities based on the deployment of *horizontal traceability relations* between requirement specifications are suggested in [2] and [69]. The focus of the latter approach is the reuse of coarse-grain requirement specifications developed for families of software systems (or parts of these specifications) expressed as structured text. This process is termed "requirements recycling" and supports the

production of a requirements specification document for a new member in a system family that shares features with existing members but may also introduce new features, or drop and modify some of the existing ones. Driven from change scenarios that incorporate feature introduction, modification and deletion requests, this approach first locates parts in existing requirement documents that refer to the required features. Then it uses *overlap*, *dependency*, and *refinement* traceability relations that involve these parts to identify other parts in the documentation that can also be recycled. The new requirement document is produced by copying and pasting the latter parts. However, while preparing the new document, the approach suggests the investigation of *overlap* relations in order to identify possibly redundant parts that should be eliminated.

5.4 Traceability for artefact understanding

A significant driver for the generation of traceability relations is to use them in order to understand the artefacts that they involve in reference to the context in which they were created, or in reference to other artefacts related to them. This is particularly important in cases where the people who need to access, understand, and maintain the artefacts are not those who contributed to their creation, a phenomenon that is typical in software maintenance.

The comprehension of source code artefacts is, for instance, one of the main objectives underpinning the generation of traceability relations between such artefacts and manual pages, requirement models [4, 38, 40] test cases, and system feature descriptions (e.g. *Software Reconnaissance technique* [71]), or design and domain analysis models [9]. The same objective of enabling code comprehension has driven the development of approaches which can trace components of programs generated by deductive synthesis (e.g. variable names, function calls) onto the specifications from which they were derived. These approaches (see [66] for example) use the formal proof that led to the generated program to construct *overlap* relations to parts of the specification that drove the derivation.

Similarly, the use of *rationalisation relations* have been extensively suggested as a means of providing explanation about the form of requirement and design artefacts [28, 47, 50, 51]. Most of the traceability environments which support the recording of such relations (e.g. PRO-ART [47], REMAP [50]) record rationale based on variants of the IBIS model [13]. According to this model, artefact construction decisions can be represented in terms of the main *issues* that were considered in the construction process, the *arguments* that were articulated for these issues, and the *positions* taken by different stakeholders with regards to these issues. A slightly different approach is advocated in *EColabor* [62]. *EColabor* supports the recording of rationale for requirements specifications articulated around questions, answers, reasons, and commitments according to the *Inquiry Cycle* model using audio and video artefacts.

EColabor provides support for fine-grain traceability relations between requirements specifications and segments of audio and video artefacts, and facilities monitoring of on-going discussions.

5.5 Empirical studies of the use of traceability

Empirical studies about the use of traceability in industrial organisations have confirmed its deployment for supporting the activities outlined in the preceding sections. They have also indicated significant differences in relevant practices, which are influenced by numerous environmental, organisational, and technical factors.

Based on the findings of a case study of 16 organisations in the US, Ramesh [49] has distinguished two different types of traceability users: the *low-end* and *high-end users*. Low-end users use traceability relations to allocate requirements to system components, inform system verification procedures, and manage changes in system development and maintenance life cycle. However, they do not view traceability as an important task in their development process and do not realise any form of "formal methodology" in their traceability practice. High-end users, on the other hand, are organisationally committed to traceability and tend to make extended use of it as part of well-defined system development policies. Thus, they capture traceability relations between artefacts created in the entire life cycle of a system and relations between artefacts and rationale. They also tend to customise the tools that they use to provide better support for their traceability practices.

As indicated in other empirical studies [5], the main factor that prohibits the wide and effective use of traceability in industrial setting is the ability to establish traceability relations that could support the required forms of analysis in cost-effective ways. The diversity of the artefacts which are generated in the software development life-cycle and the lack of interoperability between the tools that are used to construct and manage them, make the capture of traceability relations expensive and create the perception that the benefits from establishing traceability are not justified. Additional costs also arise from the need to train users to use the relevant tools and platforms. This makes traceability a cumbersome task for short-term projects [5].

Finally, it should be noted that, since the accuracy of forms of analysis that deploy traceability relations depends on the correctness and completeness of these relations and these properties cannot be guaranteed for neither manually nor automatically generated traceability relations, users tend to be reluctant to use traceability relations in such forms of analysis. Empirical studies (e.g. [37]) have confirmed this tendency and suggest that in cases where there is access to experienced software engineers, organisations rely on them for certain forms of analysis (e.g. change impact) rather than deploying recorded traceability relations, despite the cost of this approach.

6. Open Research Issues

As discussed in the preceding sections, software traceability has been the focus of numerous as well as diverse research activities in the areas of software and systems engineering over the last 15 years. Numerous approaches tackling different aspects and issues of traceability have been proposed, and tools to support the establishment, maintenance and deployment of traceability during system development have been produced. However, despite these developments, empirical studies demonstrate that current technology does not provide sufficient support for traceability and, as a consequence, traceability is not widely adopted in industrial settings unless there is a regulatory reason for doing so [5, 49, 26, 51].

In the following, we discuss the main issues, which in our view prevent the widespread use of traceability in industrial settings and should be addressed by further research. Our discussion has been based on our study of the literature, the experience that we have gained from building systems to support traceability for industrial organisations [61,78], and discussions that have taken place as part of two international workshops on traceability that we have organised over the last two years, namely TEFSE 2002 [65] and TEFSE 2003 [66].

Evidently from Sections 2 and 3, there are relatively few approaches that can automate the generation of traceability relations. With some exceptions, the relations that can be generated by these approaches do not have the strong semantic meaning required for the most important forms of analysis that can be based on traceability. Overlap relations, for instance, which can be automatically generated by existing approaches have a very general referential meaning and cannot support effectively significant forms of analysis such as change impact analysis. On the other hand, a close analysis of the reviewed approaches reveals that, with the exception of TOOR [44], none of them can support the generation of satisfiability relations which are important for analysis related to software verification. This problem becomes more significant in the case of relations that involve artefacts, which incorporate chunks of text such as requirement specifications, descriptions of rationale, and test cases. Most of the existing approaches assume that traceability relations involving artefacts of these types must be asserted manually and only a few of the reviewed approaches (i.e., [4], [40] and [61]) support the automatic generation of traceability relations involving them. Thus, as the manual assertion of traceability relations is labour intensive, traceability is rarely established unless there is a regulatory reason for doing so.

A possible way forward in the automatic generation of traceability relations could be the standardisation of vocabularies that may be used to model systems in specific application domains. Another solution along the same direction could be

based on the development of ontologies. Ontologies can provide formal specifications of common aspects of software systems and their domains (e.g. specifications of certain types of requirements in abstract forms). Thus they can be used as a starting point for building system models with the precise semantic meaning that is required for the generation of traceability relations. For instance, requirement satisfiability relations could be established by reasoning about the satisfiability of the axiomatic definition of a requirement in ontology by other system artefacts such as design specifications.

Another problem related to the generation of traceability relations is the lack of mechanisms for verifying the correctness and completeness of these relations and, therefore, establishing the necessary degree of trust that is required for deploying them for further analysis with confidence. This is mainly an issue for approaches that automate the generation of such relations. At the moment, very few of these approaches provide mechanisms through which they could be tuned so as to achieve better performance in terms of correctness and completeness. A notable exception to this phenomenon is the rule base tracer discussed in [61]. This system uses historical assessments of automatically generated traceability relations provided by the users to compute degrees of confidence in the ability of its rules to generate correct relations for new sets of artefacts. These degrees of confidence can be used to de-activate and re-activate specific traceability rules if necessary [57]. It also uses machine learning techniques for generating traceability rule that could improve the levels of completeness that it achieves [58].

Clearly the levels of correctness and completeness which are required in different settings depend on the type of the involved traceability relations and the types of analysis that they are to be deployed for. In change impact analysis, for instance, it may be more desirable to have high completeness rather than correctness rates. On the other hand, in system verification and validation correctness seems to be more important. The development of clear guidelines for making decisions about performance criteria in specific settings and mechanisms for tuning different approaches, in order to achieve better performance are issues that need to be addressed by further research. It should be noted that mechanisms for verifying the correctness and completeness of traceability relations are necessary also for systems/approaches, which assume that such relations are asserted manually. This is because evidence from our empirical investigations has indicated that it is not unlikely to have different users suggesting different sets of relations for the same set of artefacts, depending on their interpretation of the contents of these artefacts.

In the absence of sufficient support for the automatic generation and verification of traceability relations, the cost of establishing traceability in industrial settings is high. Given this, the systematic and widespread adoption of traceability as part of software development processes would certainly require clear evidence about the potential benefits from its deployment. Current empirical research fails to provide

hard quantitative evidence about these benefits and, to the best of our knowledge; methods that could be used to measure these benefits do not exist. Furthermore, although it has been argued that the adoption of 33

traceability cannot be effective unless it is tailored to the needs of specific projects and organisations [20], there is little (if any at all) methodological support for identifying these needs and using them to inform traceability adoption and deployment strategies. The development of methodological support for this purpose should try to relate traceability establishment and deployment to the objectives of the involved organisations, the existing software development strategies, and the needs of the different types of stakeholders who are envisaged to make use of traceability (e.g. managers, developers, customers, auditors etc.).

It should also be noted that the current level of tool support for traceability is one of the main reasons for its limited use in industrial settings. This is because most of the industrial tools and environments fail to provide support for all the types of artefacts that are constructed in the software development life-cycle, as well as all the types of traceability relations that may exist between these artefacts. Moreover, existing tools and environments do not interoperate with other tools, which are likely to be used in distributed and heterogeneous software development settings. Some research prototypes (e.g. EBT [10], and TraceM [55]) appear to address the interoperability problem. However, these tools do not adequately address the problem of automatic generation of traceability relations, and it is unclear whether they can achieve the data management effectiveness and robustness required in industrial settings. To this end, further research and development is required for delivering the right combination of capabilities.

7. Conclusions

In this article we have presented a roadmap of the research and practical work that has been developed to support software system traceability. To produce this roadmap, we have reviewed and presented: (a) different frameworks and classifications of traceability relations, (b) different approaches to the generation of traceability relations including manual, semi- automatic and automatic approaches, (c) different approaches to the representation, recording, and maintenance of traceability relations that underpin the architectural design and implementation of traceability tools and environments, and (d) different ways of deploying traceability relations in the software development process. Our review of the field has also identified issues that require further investigation by both the research and industrial communities which are also presented in the paper.

Our view is that the establishment and effective deployment of software traceability is very significant in the software development life cycle, and that existing approaches have made significant contributions to various aspects of

traceability. However, it has to be appreciated that the provision of adequate support for traceability is not an easy task and given the current state of the art in this field it cannot be claimed that holistic and effective support for traceability is available. As we discussed in Section 6, there is still a significant number of issues of traceability which are open to further research and need to be addressed adequately before traceability can be widely adopted in industrial settings and benefit system development processes.

References

1. Alexander I., "Towards Automatic Traceability in Industrial Practice", *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2002)*, Edinburgh, UK, September 2002.
2. Alexander I., "SemiAutomatic Tracing of Requirement Versions to Use Cases – Experience and Challenges, *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2003)*, Canada, October 2003.
3. Alexander I, Kiedaisch F, "Towards Recyclable System Requirements", *Proceedings of the 9th Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, 9- 17, 2002
4. Antoniol G., Canfora G., Casazza G., De Lucia A., Merlo E., "Recovering Traceability Links between Code and Documentation", *IEEE Transactions on Software Engineering*, 28(10), 970-983, October 2002
5. Arkley P., Mason P., Riddle S., "Position Paper: Enabling Traceability", *Proceedings of 1st International Workshop on Traceability in Emerging Forms of Software Engineering*, 61-65, available from: http://www.soi.city.ac.uk/~zisman/traceworkshop.html
6. Bayer J., Widen T., "Introducing Traceability to Product Lines", *Proceedings of the Software Product Family Engineering: 4th International Workshop, PFE 2002*, Bilbao, Spain, Lecture Notes in Computer Science, Springer-Verlag, ISSN:0302-9743.
7. Bayet J., Flege O., Knauber P., Laqua R., Muthig D., Schmid K., Widen T., DeBaud J.M., "PuLSE: A methodology to develop software product lines", in *Symposium on Software Reusability*, May 1999.
8. Bianchi A, Fasolino A.R, Vissagio G, "An Exploratory Case Study of the Maintenance Effectiveness of Traceability Models" *Proceedings of 8th International Workshop on Program Comprehension (IWPC'00)*, 149-159, Limerick, Ireland, June 2000
9. Boldyreff C., Burd E.L., Hather R.M., Munro M., Younger E.J., "Greater Understanding Through Maintainer Driven Traceability", *Proceedings of the 4th International Workshop on Program Comprehension (WPC'96)*, 1996.
10. Cleland-Huang J., Chang C., Wise J., "Supporting Event Based Traceability through High-Level Recognition of Change Events", *Proceedings of IEEE COMPSAC Conference*, Oxford, England, August 2002
11. Cleland-Huang J., Schmelzer D., "Dynamic Tracing Non-Functional Requirements through Design patter Invariants", *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2003)*, Canada, October, 2003.
12. Cleland-Huang J., Chang C.K., Sethi G., Javvaji K., Hu H., Xia J., "Automating Speculative Queries through Event-based Requirements Traceability", *Proceedings of the IEEE Joint International Requirements Engineering Conference*, Essen, Germany, September 2002.

13. Conklin J, Begeman M, "gIBIS: A Hypertext Tool for Exploratory Policy Discussion", *ACM Transactions on Office Information Systems*, 6, 303-331, October 1988
14. Constantopoulos P, Jarke M, Mylopoulos Y, Vassiliou Y, "The Software Information Base: A Server for Reuse", *VLDB Journal*, 4(1), 1-43, 1995
15. Cyre W., Thakar A., "Generating Validation Feedback for Automatic Interpretation of Requirements", *Formal Methods in System Design*, Kluwer, 1997.
16. Cysneiros G., Zisman A., Spanoudakis G., "A Traceability Approach for *i** and UML Models", *Proceedings of 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems - ICSE 2003*, May 2003
17. CORE, http://www.vtcorp.com
18. DeRose S., Maler E., Orchard D., "XML Linking Language (XLink)", version 1.0, http://www.w3.org/TR/2000/REC-xlink-20010627/, World Wide Web Consortium.
19. Dick J., "Rich Traceability", *Proceedings of the 1st International Workshop on Traceability for Emerging forms of Software Engineering (TEFSE'02)*, Edinburgh, UK, September 2002.
20. Dogmes R., Pohl K., "Adopting Traceability Environments to Project-Specific Needs", *Communications of the ACM*, 41(12), 55-62, 1998.
21. Easterbrook S, Callahan J, Wiels V, "V & V through Inconsistency Tracking and Analysis", *Proceedings of the 9th International Workshop on Software Specification and Design*, 43-51 ,1998
22. Egyed A., "A Scenario-Driven Approach to Trace Dependency Analysis", *IEEE Transactions on Software Engineering*, Vol. 9, No. 2, February, 2003.
23. Egyed A., Gruenbacher P., "Automatic Requirements Traceability: Beyond the Record and Replay paradigm", *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE)*, Edinburgh, UK, September, 2002.
24. Fiutem R, Antoniol G., "Identifying Design-Code Inconsistencies in Object-Oriented Software: a Case Study", *Proceedings of International Conference on Software Maintenanc*e, 94- 103, March 1998.
25. Gamma E., Heml R., Johnson R., and Vlissides J., " Design Patters: Elements of Reusable Object-Oriented Software", *Addison-Wesley Professional Computer Series*, 1st Edition, ISBN 0201633612, 1995.
26. Gotel O., Finkelstein A., "An Analysis of the Requirements Traceability Problem", *Proceedings of the 1st International Conference in Requirements Engineering*, 94-101, 1994.
27. Gotel O., Finkelstein A., "Contribution Structures", *Proceedings of 2nd International Symposium on Requirements Engineering, (RE '95)*, 100-107, 1995.
28. Haumer P, Pohl K, Weidenhaupt K, Jarke M, "Improving Reviews by Extended Traceability", *Proceedings of 32nd International Conference on System Sciences*, 1999
29. Hayes J.H., Dekhtyar A., Osborne J., "Improving Requirements Tracing via Information Retrieval", *Proceedings of the 11th IEEE International Requirements Engineering Conference*, Monterey Bay, 2003.
30. Integrated Chipware, RTM, www.chipware.com
31. Kaindl H., "The Missing Link in Requirements Engineering", *Software Engineering Notes*, June 1992, pp. 498-510
32. Kozlenkov A., Zisman A., "Are their Design Specifications Consistent with our Requirements?", *Proceedings of IEEE Joint International Requirements Engineering Conference - RE'02*, Essen, September 2002.
33. Lavazza L, Valetto G, "Requirements-based Estimation of Change Costs", *Empirical Software Engineering - An International Journal*, 5(3), November 2000

34. Lee C., Guadagno L., Jia X., "An Agile Approach to Capturing Requirements Traceability", *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2003)*, Canada, October, 2003.
35. Letelier P., "A Framework for Requirements Traceability in UML-based Projects", *Proceedings of the 1st International Workshop on Traceability for Emerging Forms of Software Engineering (TEFSE'02)*, Edinburgh, UK, September 2002.
36. Lindval M., Sandahl K., "Practical Implications of Traceability", *Software Practice and Experience*, vol. 26, no. 10, 1996, pp 1161-1180.
37. Lindvall M, Sandahl K, "Traceability Aspects of Impact Analysis in Object Oriented Systems", *Software Maintenance: Research and Practice*, 10(1), 37-57, 1998
38. Maletic J.I., Marcus A, "Supporting Program Comprehension Using Semantic and Structural Information", *Proceedings of 23rd International Conference on Software Engineering*, 103-112, Toronto, Canada, 2001.
39. Maletic J.I., Munson E.V., Marcus A., Nguyen T.N., "Using a Hypertext Model for Traceability Link Conformance Analysis", *Proceedings of the 2nd International Workshop on Traceability for Emerging Forms of Software Engineering (TEFSE'03)*, Canada, October 2003.
40. Marcus A., Maletic J.I., "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing", *Proceedings of 25th International Conference Software Engineering*, 2003
41. Mohan K., Ramesh B., "Managing variability with Traceability in product and Service Families", *Proceedings of the 35th Hawaii International Conference on System Sciences*, IEEE, 2002.
42. NASA, "Preferred Reliability Practices: Independent Verification and Validation of Embedded Software", *Practice No. PD-ED-1228*, Marshal Space Flight Centre
43. Perez-Minana E, Trew T, Krause P, "Issues on the Composability of Requirements Specifications for a Product Family", *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering*, 47-53, September 2002
44. Pinheiro F., Goguen J., "An Object-Oriented Tool for Tracing Requirements", *IEEE Software*, 52-64, March 1996.
45. Pinheiro F., "Formal and Informal Aspects of Requirements Tracing", *Proceedings of 3rd Workshop on Requirements Engineering (III WER)*, Rio de Janeiro, Brazil, 2000
46. Pohl K., *Process-Centered Requirements Engineering*. John Wiley Research Science Press, 1996.
47. Pohl K, PRO-ART: Enabling Requirements Pre-Traceability, *Proceedings of the 2nd IEEE International. Conference on Requirements Engineering (ICRE 1996)*, 1996
48. Pohl K. et al., "Product Family Development", *Dagstuhl Seminar Report No.304*, http://www.dagstul.de/01161/report, 2001.
49. Ramesh B., "Factors Influencing Requirements Traceability Practice", *Communications of the ACM*, 41(12), pp. 37-44, 1998.
50. Ramesh B., Dhar V., "Supporting Systems Development Using Knowledge Captured During Requirements Engineering", *IEEE Transactions in Software Engineering*, June 1992, 498-510, 1992.
51. Ramesh B., Jarke M., "Towards Reference Models for Requirements Traceability", *IEEE Transactions in Software Engineering*, 27(1), 58-93, 2001.
52. RDT, http://www.igatech.com/rdt/index.html
53. Riebisch M., Philippow I., "Evolution of Product Lines Using Traceability", *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*, in conjunction with OOPSLA 2001, Tampa Bay, Florida, USA, October 2001.

54. Savolainen J., "Tools for Design Rationale Documentation in the Development of a Product Family", *Proceedings of 1st Working IFIP Conference on Software Architecture*, San Antonio, Texas, 1999.
55. Sherba S.A., Anderson K.M., Faisal M., "A Framework for mapping Traceability Relationships", *Proceedings of the 2nd International Workshop on Traceability for Emerging forms of Software Engineering (TEFSE 2003)*, Montreal, Canada, September, 2003.
56. Song X, Hasling B, Mangla G, Sherman B, "Lessons Learned from Building a Web-Based Requirements Tracing System", *Proceedings of 3rd International Conference on Requirements Engineering*, 41-50, 1998
57. Spanoudakis G., "Plausible and Adaptive Requirement Traceability Structures", *Proceedings of the 14th International Conference in Software Engineering and Knowledge Engineering*, Ischia, Italy, pp. 135-142, 2002
58. Spanoudakis G., Avilla Garcez A., Zisman A., "Revising Rules to Capture Requirements Traceability Relations: A Machine Learning Approach", *Proceedings of the 15th International Conference in Software Engineering and Knowledge Engineering (SEKE 2003)*, San Francisco, USA, July 2003.
59. Spanoudakis G, Finkelstein A, Till D, "Overlaps in Requirements Engineering", *Automated Software Engineering Journal*, 6(2), 171-198, 1999
60. Spanoudakis G., Kim H., "Supporting the Reconciliation of Models of Object Behaviour", *International Journal of Software and Systems Modelling*, 2004 (to appear)
61. Spanoudakis G., Zisman A., Perez-Minana E., Krause P., "Rule-Based Generation of Requirements Traceability Relations", *Journal of Systems and Software*, 72(2), pp. 105-127, 2004
62. Strens M, Sugden R., "Change Analysis: A Step towards Meeting the Challenge of Changing Requirements", *Proceedings of the IEEE Symposium and Workshop on Engineering of Computer-Based Systems*, Fredrichshafen, Germany, March 1996
63. Takahashi K, Potts C, Kumar V, Ota K, Smith J, " Hypermedia Support for Collaboration in Requirements Analysis", *Proceedings of 2nd International Conference on Requirements Engineering*, 31- 40, 1996
64. Teleologic, Teleologic DOORS, www.teleologic.com/products/doors
65. Spanoudakis G, Zisman A, "Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering (organised in conjunction with the 17th IEEE International Conference on Automated Software Engineering)", (Eds) Spanoudakis G., Zisman A., Perez-Minana E., September 2002.
66. Spanoudakis G, Zisman A, "Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (organised in conjunction with the 18th IEEE International Conference on Automated Software Engineering)", (Eds) Spanoudakis G., Zisman A., September 2003.
67. Van Baalen J., Robinson P., Lowry M., Pressburger T., "Explaining Synthesized Software", *Proceedings of the 13th IEEE Conference on Automated Software Engineering*, 240-248, October 1998
68. Von Knethen A., "Automatic Change Support Based on a Trace Model", *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'02)*, Edinburgh, September 2002.
69. Von Knethen A., Paech B., Kiedaisch F., Houdek F., "Systematic Requirements Recycling through Abstraction and Traceability", *Proceedings of the IEEE International Requirements Engineering Conference*, Germany, September 2002.
70. Watkins R, Neal M, "Why and How of Requirements Tracing", *IEEE Software*, 104-106, July 1994

71. Whitehead E., "An Architectural Model for Application Integration in Open Hypermedia Environments", *Proceeding of the 8th ACM Conference on Hypertext*, 1-12, April 1997.
72. Wilde N, Casey C, " Early Field Experience with the Software Reconnaissance Technique for Program Comprehension", *Proceedings of the 1996 International Conference on Software Maintenance (ICSM '96)*, 312-318, 1996
73. Xlinkit. http://www.systemwire.com/xlinkit.
74. Xu P., Ramesh B., "Supporting Workflow management Systems with Traceability", *Proceedings of the 35th Hawaii International Conference on System Sciences*, IEEE, 2002.
75. Zowghi D., Offen R., "A Logical Framework for Modelling and Reasoning about the Evolution of Requirements", *Proceedings of 3rd International Symposium on Requirements Engineering*, Annapolis, MD, January 1997
76. Zisman A., Emmerich W., and Finkelstein A.. "Using XML to Build Consistency Rules for Distributed Specifications", *Proceedings of 10th International Workshop on Software Specification and Design - IWSSD-10*, San Diego, USA, November, 2000.
77. Zisman A., Spanoudakis G., Perez-Minana E., Krause P., "Towards a Traceability Approach for Product Families Requirements", *Proceedings of 3rd ICSE Workshop on Software Product Lines: Economics, Architectures, and Implications*, Orlando, USA, May 2002
78. Zisman A., Spanoudakis G., Perez-Minana E., Krause P., "Tracing Software Requirements Artefacts", *Proceedings of the 2003 International Conference on Software Engineering Research and Practice (SERP'03)*, Las Vegas, Nevada, USA, June 2003.
79. Zisman A., Kozlenkov A., "Consistency Management of UML Specifications", *Proceedings of 4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03)*, Lübeck, Germany, October, 2003.