Traceability in Collaborative Systems Development from Lifecycle Perspective
– A Position Paper

Darijus Strašunskas
Dept. of Computer and Information Science, Norwegian Univ. of Science and Technology
Sem Sælands vei 7-9, NO-7491 Trondheim, Norway
dstrasun@idi.ntnu.no

Abstract

The aim of this position paper is to discuss the features of state-of-the-art and outstanding issues of the traceability between product fragments in collaborative system development.

A lot of research has been done in the pre-traceability area. Recently, researchers' attitudes towards inter-relation of requirements and architecture elements have increased. Several approaches to tackle this problem have been proposed. Nevertheless, to the author's knowledge, the solution for traceability between various product fragments through the lifetime of the system does not exist.

Central repository for the traceability relationship and distributed repositories for the model fragments storage and interchange between developers are proposed. Usage of ontology is proposed to interrelate different product fragments and establish the traceability relations between them.

1. Introduction

Traceability and its relevance to the software development process are well described in the literature (e.g., [8], [28]). Especially much research is done in pre-requirements traceability to capture rationale (e.g., [9], [22], [24]), but there is the lack of traceability through the entire system development process.

An information system is viewed as a product composed of product fragments, which are again compositions of sub-fragments:

– *Model fragments* – are sub-models of a conceptual model of the information system being under development. Only the semantic content of the model is stored, not diagram layout information.

– *Diagrams* – stores layout information of the conceptual model view. Diagrams may exist in several different layout versions without affecting the conceptual content.

– *Code fragments* – are code modules (files).

– *Document fragments* – are pieces of the documentation of the models and code fragments.

These four fragments types are described differently. Model structure depends on the conceptual model languages used, diagram structure reflects the basics of visual languages, code structure is expressed in programming language and document structure reflects common document architectures.

Every decision and rationale behind it should be captured and traced during system development. The traceability of the history of a product is a prerequisite for managing evolution of product. Capturing and maintaining traces from requirements to implementation and vice versa have long ago been acknowledged as one of essential systems development activities [19].

There are several aspects that make the traceability between requirements and later fragments of software development (design, architecture, code fragments) problematic. First, conceptual distance between two worlds: human (requirements are captured in natural language) and technical (entities are specified in formal method). Second, it is difficult to maintain the consistency and traceability between different fragments since single requirement could map multiple architectural and design concerns derived from it. Contrarily, architectural component could have few relations to various requirements. And third, large systems should satisfy hundreds, even thousands of requirements, and this makes second issue even more complicated.

This paper is the statement about research in progress within the area of traceability between various fragments (documents, models, model elements, code) in a collaborative software development throughout entire system lifecycle. The research has so far mainly been built on an extensive literature study and survey of the requirements management tools.

The paper is organized as follows: first, the traceability issues and existing approaches are discussed in light of system development lifecycle. Next section discusses the aspects of traceability in a collaborative work and lists deficiencies of CASE tools. The research issues and a vision of approach to tackle them are enlightened in section 4. Finally, conclusions are presented.

2. Traceability from a Life Cycle Perspective

Gotel and Finkelstein [8] define requirements traceability as "the ability to describe and follow the life of a requirement in both forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)".

Software systems always evolve as the environment and stakeholders' requirements change. Therefore, managing change is a fundamental activity not only in requirements engineering (RE), but also in overall system development.

Neglecting traceability or capturing insufficient traces could decrease system quality, and extend development time. It is important to trace which model element is affected by change of requirement. And, vice versa, how change of model element corresponds to defined requirements, and consequently how will the system functionality be affected.

The systems specification is developed in an iterating manner. Concurrent Engineering requires the cooperation of people coming from different phases of the engineering process. Traceability between the different views (diagrams), which exist in such cross-functional teams, is essential for enabling mutual understanding. Moreover, the different views must be related to each other and must be presented in a suitable way to support finding and resolving of inconsistencies, conflicts, and different opportunities.

One of the solutions to the problem of traceability is the hyper linkage of documents from different phases of development. If these documents are suitably annotated, they can provide a meaningful design history throughout the development lifecycle [3] and increase the browsing capabilities. This could be done mainly manually or semi-automatically using linguistics technique. Cerbah and Euzenat [2] have implemented system that generates class hierarchies out of textual requirements specifications and establishes traceability between models and texts through terminology. The authors have not explicitly stated, but the same technique could be use to relate the documentation.

The CBSP (Component, Bus, System, Property) approach [11] deals with refinement of requirements to initial architecture, as requirements may explicitly or implicitly contain information relevant to the system's architecture. A Scenario and Meta-Model Based Approach [23] integrates requirements and architecture information by defining orthogonal meta-models, which define the concepts and relations about information to be recorded during system development. It is based on six meta-models. Meta-models are used to structure requirements information and interrelate structured information.

Murphy et al. presents formal technique for abstraction of source code information to match higher-level model elements [17]. The abstractions for consistency checking between model and code, but not interrelation between different model elements, are used.

Approach by Hall et al. [12] uses extended problem frames [14], which allow architectural structures, services and artefacts to be considered as part of the problem domain.

Jviews by Grundy et al. [10] uses class-like diagrams and other design-level constructs. They focus on lower-level design and address a series of view integration problems in that level.

ViewPoints [6] framework by Finkelstein et al. presents some views and corresponding rules to identify inconsistencies within and between them. Framework provides mechanisms for detecting, classifying and resolving inconsistencies. ViewPoint resorts to a more formal specification of requirements. ViewPoint support for design and architecture type diagrams is not explicitly stated, although they state that some exists.

Comparison of those approaches ([2], [6], [10], [11], [12], [17], [23]) for relating different views in system development is summarized in table 1. Approaches are compared in context of their coverage of lifecycle phases. Lifecycle phases are split into requirements, architecture, design, coding and system documentation. System development lifecycle is assumed to be an iterative process, not waterfall-like.

Table 1. Comparison of approaches from lifecycle perspective

Phases of lifecycle / Aproaches	Requirements	Architecture	Design	Coding	Documentation
Linguistics approach [2]	■	□	◪	□	■
CBSP [11]	■	■	□	□	□
Scenario & Meta-Model Based [23]	■	■	□	□	□
Murphy et al. [17]	□	□	◪	■	□
Approach using Problem Frames [12]	■	■	□	□	□
Jviews [10]	□	□	■	□	□
ViewPoints [6]	■	◪	◪	□	□

■ - high ◪ - average □ - none

The following ratings were assigned: *high*, *average*, *none*. The average and *high* ratings are given when some support or extensive support is available. A *none* rating is

assigned when the criteria are not satisfied by the approach or are insignificant.

In general, research is done in the area of traceability between two different phases of software development (either between requirements and rationale behind them, or requirements and architecture, or source code and model elements (see table 1)). Most of the investigated approaches do not take into consideration documentation phase of system lifecycle (user manuals, blueprints and etc.). Consistent update of the system documentation is one of the prerequisites for its maintainability. Of course, documentation could be considered to be similar to requirements specification documents (as content is mainly in natural language). Nevertheless, most of approaches at requirements level deals with semi-formal specification (scenarios, use case) and they hardly consider plain language documents. This raises the first question: is it possible to maintain backward and forward traceability between various fragments through the lifetime of the system?

Desirable types of traceability relationships in an iterative system development are depicted in fig. 1. Interrelationships between related fragments of different abstraction levels inside of every stage should be maintained as well as relationships between fragments of different lifecycle stages. It is important to control more detail representation level conformity to the problem domain, as abstraction mechanisms are used to simplify picture of the system and problem domain, which is further used for communication with the non-technical users.

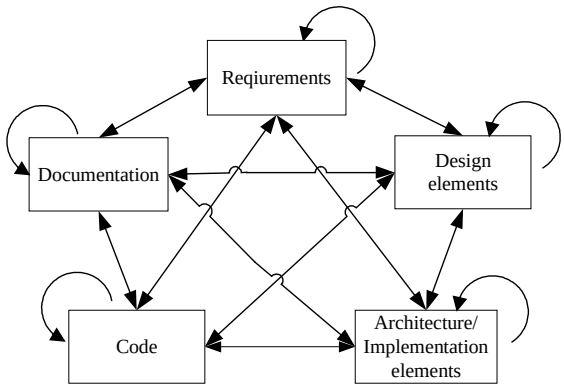

Figure 1. Desirable traceability links throughout system lifecycle.

The necessity of the link between requirements and code is not so obvious, but could be useful in an evolutionary prototyping, when a prototype is used to extract the requirements from IKIWISI (I'll know it when I see it) users.

3. Traceability in Environment for Collaborative System Development

3.1. Deficiencies of the CASE tools

Large, net-wide collaborative system engineering projects become a reality by the recent growing communication and data exchange possibilities brought about by the fast-growing Internet-related developments. These projects share system models among users, tools and repositories beyond geographical boundaries, which could result in significant savings in time and effort. Models that have been created or processed by one tool could serve as the initial step for the next tool. Shared models could also enable the composition and integration of the most appropriate tools for a given activity. Especially, as many CASE (Computer Aided System Engineering) tools are developed as "time sharing" systems [5]: each developer is given the feeling of being the "only user" of the system.

In addition, the format of the product objects are often controlled by strict consistency checks, making the evolution of product objects from informal ideas to formal constructs difficult. As a consequence, CASE tools might be reduced to tools for documenting products that are developed outside the CASE tools. That is not what CASE tools are built for. CASE tools in the best case confine themselves to offering a central repository where information about the product can be accessed regardless of geographical location.

Weaknesses in CASE tool support could be divided into the following aspects [20]:

- Lack of mechanism for integrating sets of methods while maintaining consistency between various models,
- Lack of support of multiple users to create, modify and delete sets of partly overlapping model instances,
- Inadequate catering for multiple representational requirements raging from fully diagrammatic to fully textual or matrix representation.
- Failure to provide consistent mapping mechanism between different representational paradigms.
- Lack of flexibility and evolvability in method support ranging from syntactic variation in methods to crafting totally new method components.
- Insufficient catering for different information-related needs of a diverse set of stakeholders.

The cost-efficient sharing of modelling information between heterogeneous tools and repositories requires the adoption of a standard for an industry-wide model interchange format. However, no such standard yet exists. There are several model interchange formats: XIF [16], XMI [21], SPOOL [26], UXF [27]. Product fragments are stored and interchanged through a repository. The repository may be useful during various phases in the lifecycle of an information system (for instance, as a basis for various decisions or reuse).

3.2. Specific Aspects of Collaborative Environment

Each engineer develops its own product (fragment) using his/ her preferred representation. At a certain time, the products developed in parallel must be integrated; discrepancies and similarities must be detected through the communication and conversation among the people involved. Changes to the products have to be made according to unresolved discrepancies. Fundamentally, concurrent engineering relies on the capability to merge pieces of work done in a concurrent way on the same object. Merging objects is thus a central issue. Trace information should be captured and traceability between related fragments should be re-established.

Farshchian in [5] emphasizes the list of requirements for product development environments; some of them (relevant to this research) are listed below:

- Flexible access to the product – a product development environment should provide flexible mechanisms for accessing and updating the product.
- Unrestricted product object types – a product development environment should allow the developers to share any type of object that they might find useful for supporting their cooperation.
- Unrestricted relation types – a product development environment should allow the developers to create any type of relation between any two objects of product.
- Incremental product refinement – a product development environment should provide the developers with flexible mechanisms for incrementally refining the product. The developers should be allowed to start with vague products, and to refine them into more complete and formal ones.
- Support for boundary objects – a product development environment should allow the developers to view the product from different perspectives. The environment should in addition support a global view of the product.
- Active delivery of information – a product development environment should take an active part in delivering necessary information to the developers. In particular information about changes to the shared product should be delivered continuously to the interested developers.

Traceability technique integrated with the version control and configuration management could facilitate management of the composition of product fragments consisting of interrelated various model fragments, code fragments and documents:

Merging. Merging allows multiple versions to be joined together, producing a new version representing the union of the actions taken from previous version. Fully automatic mergers are difficult to implement due to semantic considerations during the resolution of conflicting changes. In case the change has been applied in only one version, this change can be incorporated automatically; otherwise, a conflict that can be resolved automatically or manually is detected. Merging should handle various levels of granularity.

Access control should be applied at different level of granularity, for instance, model, object, and attributes.

Version management. The collaborative CASE tools should keep track of changes in different working modes [15]:

- Multiple developers are working on a single common version simultaneously;
- Developers are working individually on their local versions;
- Both cases: some developers collaborating synchronously, others working individually.

In first case the system should track and resolve multiple edits by different developers in the same fragment. Changes could occur simultaneously or sequentially in collaborative session. Considering second case – there will be a number of current versions – the system should provide and manage awareness of their existence and dependencies among them. Third case incorporates previous both – asynchronous and synchronous developing.

In [13] it is stated five different properties of configurations must be maintained during systems development:

Authorized. A configuration is authorized if and only if all of its objects are defined as being authorized. An authorized configuration is also called a baseline.

Consistent. The objects of a consistent configuration have to be consistent, both relative to each other and as individual components. A configuration remains invalid if it is not checked or if the consistency checker detects inconsistencies.

Latest. Each object of a latest configuration must be the latest version of its own family.

Owner. The property owner constrains the configuration to contain only object versions owned by a given user. Authorized versions are considered as owned by all project group members.

Project-wide. A project-wide configuration must include one object version from every family within project.

Three configurations are important: 1) the *latest project-wide*; 2) *latest consistent*; 3) *latest authorized*. The latest project-wide baseline reflects all new developments in the project. Latest baselines are not supposed to be consistent, but all new ideas since last logon are detected by inspecting this configuration. Latest consistent baselines represent the most recent stable work. These configurations are the candidates for authorization. There may exist several latest, authorized configurations,

but usually exists one. This is the last configuration, which the group agree on, and it forms the official basis of all subsequent work. Traceability technique between different versions of the product fragments with incorporated configuration management facilitates extraction of relevant configuration.

4. Research Issues and Future Work

From a development perspective four particularly important areas have been identified: traceability from requirements to implementation elements through design, interchange of product fragments, configuration management and concurrent work.

Internet technologies (XML, RDF) contribute the model interchange formats for trace information by capturing and exchanging through repositories, and enabling semantic interoperability. Model interchange formats should support any inter-model consistency check, and semantic validation. They should not be used only for static model transfer between environments.

4.1. Research areas

Considering perspectives mentioned in the chapters above, two main types of traceability should be maintained in an ideal cooperative software development:

- Traceability from requirements to implementation – most difficult to implement as various diagrams could be used in the development project;
- Traceability between versions and configuration. The information about the essence of new version development and the rationale behind it should be captured. This is similar to the pre-traceability task – to capture the rationale behind requirements.

The question is: *What kind of trace information is possible to capture?* It could be answered over the time when understanding of the domain and solution increases, as currently there is impossible to capture all trace information.

While work in software architectures has concentrated on how to express software architectures and reason about their behavioural properties, there is still an open question about how to analyse what impact a particular architectural choice has on the ability to satisfy current and future requirements, and variations in requirements across a product family [7].

The main research question is:

How can we effectively and consistently integrate changes of development objects in different levels of granularity throughout lifecycle of geographically distributed cooperative product development?

4.2. Future Work

Concurrent engineering changes old practice, when all the required objects were locked during the whole change/ modification activity. Each software engineer should have direct access to all needed objects. But changed version should be kept with access forbidden for other developers during modification, because the state of fragment is inconsistent in a modification phase. If *n* engineers change the same object concurrently, this object should have *n+1* different copies [4]. It means that each developer needs the private copies of fragments. On the other hand, the colleagues know that other changes possibly are done on the same fragments/ objects and want to be incorporated when relevant. During the development all relevant information should be captured, traced and available for all project participants, depending on access rights.

The requirement for consistent evolving product structures introduces high complexity in the software configuration management systems. Furthermore, this requirement restricts cooperative work on the same structure since cooperative work necessarily means that the product structure is in a state of inconsistency most of the time [1]. “Lazy” consistency [18] could be introduced to avoid disturbance of the developers’ creativity and to satisfy the requirements for the *incremental product refinement*. This approach favours software development architectures where impending or proposed changes, as well as changes that have already occurred, are announced. This allows the consistency requirements of a system to be “lazily” maintained as it evolves. Lazy consistency maintenance supports activities such as negotiation and other organizational protocols that support the resolution of conflicts and collisions of changes made by different developers.

In figure 2 an architectural proposal for traceability management in distributed cooperative system development is depicted. The central repository is used for storage of traceability links; the model fragments are stored on local workstations in shared repositories. Group awareness and access control techniques are to be used to manage collaborative work of developers. Version control and configuration management techniques are to be used on the common repository side for the traceability relations’ version control and update. These techniques are also to be used on the workstations side for the configuration management of product fragments.

Traceability relations should be based on semantic interoperability of related fragments. Every fragment has knowledge of the developer that s/he represented in. Semantic is the relationship between the fragment and the meaning (knowledge) it possesses. In order to establish the relation between the fragments, an ontology of what the fragments are about should be defined. So, the repository for traceability relations (depicted in fig. 2)

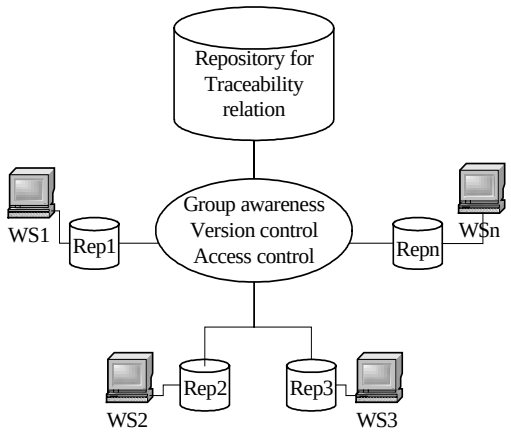

Figure 2. Deployment of traceability management in collaborative development.

should have meta-model as pre-defined ontology. Ontology would facilitate the usage of *unrestricted* product objects and relation types.

5. Conclusions

A transition between various product fragments is still unsolved problem, despite of several methods for linking requirements to architecture components (e.g., [11], [23]).

There are very few empirical studies focused on how organizations actually manage the different types of traceability. More research is required on how organizations actually capture the trace information. The real challenge is not only to investigate and propose solutions for those problems, but it is rather to provide solutions, which outweigh the cost of implementing and using them. Because the manual mapping and hyper linkage establishing are not the solutions that could be enthusiastically accepted by industry, product fragments (natural language, models and code) should be woven into a coherent whole.

It is not trivial to generate and validate trace information, but traceability could help the company:

- To ensure completeness: The user or a program can easily identify the requirements which are not satisfied by the system by following traceability links;

- To propagate the changes: At any time in the development process, traceability information allows to find out the elements impacted by changes. For instance, it could be possible to evaluate the impact on the software design and implementation of a late change in the initial customer requirements.

Phases in system development and their output are so different (one of the reasons – various modelling techniques are used) that to develop multipurpose traceability method is hardly possible. By now it is possible to develop solutions for separate modelling languages as UML [25], PPP [29] by defining the common ontology for the semantic interrelation of fragments.

6. References

[1] R. Andersen, *A Configuration Management Approach for Supporting Cooperative Information System Development*, PhD thesis, NTH, Trondheim, Norway, 1994

[2] F. Cerbah, J. Euzenat, "Traceability between models and texts through terminology", *Data and Knowledge Engineering*, Elsevier Science Publishers, 2001 38 (1), pp. 31-43.

[3] J. Corriveau, C. Hayashi, "A Strategy for Realizing Traceability in an Object-Oriented Design Environment", *Proceedings of Computer Aided Systems Theory - CAST'94*, 4th International Workshop, Ottawa, Canada, May 1994.

[4] J. Estublier, "Objects Control for Software Configuration Management", In K.R. Dittrich, A. Geppert, M.C. Norrie (Eds.): *Advanced Information Systems Engineering, Proceedings of thirteenth conference CAiSE 2001*, LNCS 2068, Springer-Verlag, Interlaken, Switzerland, June 2001, pp. 359-373.

[5] B.A. Farshchian, *A Framework for Supporting Shared Interaction in Distributed Product Development Projects*, PhD thesis, NTNU, Trondheim, Norway, 2001.

[6] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, M. Goedicke, "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development", *International Journal on Software Engineering and Knowledge Engineering*, World Scientific Publishing, march 1991, pp. 31-58.

[7] D. Garlan, "Software Architecture: A Roadmap", In A. Finkelstein (ed.), The Future of Software Engineering, Special Issue 22nd International Conference on Software Engineering, ACM Press (2000)

[8] O.C.Z. Gotel, A.C.W. Finkelstein, "An Analysis of the Requirements Traceability Problem", In *Proceeding of the 1st International Conference on Requirements Engineering* (ICRE'94), IEEE Computer Society Press, Colorado Springs, Colorado, USA, April 1994, pp. 94–102.

[9] O.C.Z. Gotel, A.C.W. Finkelstein, "Contribution Structures", In *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering* (RE'95), IEEE Computer Society Press, York, U.K., March 1995, pp. 100-107.

[10] J. Grundy, J. Hosking, W. Mugridge, "Supporting flexible consistency management via discrete change description propagation", *Software Practice and Experience*, John Wiley & Sons, 1996 26(9), pp. 1053-1083.

[11] P. Grünbacher, A. Egyed, and N. Medvidovic, "Reconciling Software Requirements and Architectures - The CBSP Approach", In *Proceedings of the 5th IEEE International*

Symposium on Requirements Engineering (RE'01), Springer-Verlag, Toronto, Canada, 2001, pp. 202-211.

[12] J.G. Hall, M. Jackson, R.C. Laney, B.A. Nuseibeh, L. Rapanotti, "Relating Software Requirements and Architectures using Problem Frames", (to appear in) *Proceedings of IEEE International Requirements Engineering Conference* (RE'02), Essen, Germany, September 2002.

[13] T.R. Henriksen, A.D. Fidjestøl, A.B. Aubert, "PPP Repository Management", *Unpublished NTNU report*, Trondheim, Norway, 15th October 1997

[14] M. Jackson, *Problem Frames*, ACM Press Book, Addison-Wesley, 2001.

[15] B.G. Lee, N.H. Narayanan, K.H. Chang, "An integrated approach to distributed version management and role-based access control in computer supported collaborative writing", *The Journal of System and Software*, Elsevier Science Publishers, 2001 59 (2), pp.119-134.

[16] Microsoft Corp., "Repository SDK 2.1b Documentation: XML Interchange Format (XIF)", Microsoft Corp., Redmond, USA, May 1999.

[17] G.C. Murphy, D. Notkin, K. Sullivan, "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models", In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, New York, NY, 1995, pp.18-28.

[18] K. Narayanaswamy, N. Goldman, ""Lazy" Consistency: A Basis for Cooperative Software Development", In *Proceedings of International Conference on Computer-Supported Cooperative Work* (CSCW'92), Toronto, Ontario, Canada, November 1992, pp.257-264.

[19] B. Nuseibeh, S. Easterbrook, "Requirements Engineering: A Roadmap" In A. Finkelstein (ed.), *The Future of Software Engineering*, Special Issue 22nd International Conference on Software Engineering, ACM Press, 2000.

[20] S. Kelly, K. Lyytinen, M. Rossi, "MetaEdit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment", In *Proceedings of CAiSE'96*, Heraklion, Greece, May 1996.

[21] OMG, "XML Metadata Interchange (XMI)", Document ad/98-10-05, October 1998.

[22] F.A.C. Pinheiro, J.A. Goguen, "An Object-Oriented Tool for Tracing Requirements", *IEEE Software*, 1996 13(2), pp. 52-64.

[23] K. Pohl, M. Brandenburg, A. Gülich, "Integrating Requirement and Architecture Information: A Scenario and Meta-Model Based Approach", In *Proceedings of the Seventh International Workshop on Requirements Engineering: Foundation for Software Quality* (REFSQ'01), Interlaken, Switzerland, 2001.

[24] K. Pohl, "PRO-ART: Enabling Requirements Pre-Traceability", In *Proceedings of the Second International Conference on Requirements Engineering* (ICSE '96), Colorado, USA, 1996, pp. 76-85.

[25] Rational Software Corp., "Complete UML 1.4 specification", 2001.

[26] G. St-Denis, R. Schauer, R.K. Keller, "Selecting a Model Interchange Format The SPOOL Case Study", In *IEEE Proceedings of the 33rd Annual Hawaii International Conference On System Sciences*, Maui, Hawaii, 2000.

[27] J. Suzuki, Y. Yamamoto, "Managing the Software Design Documents with XML", In *Proceedings of the Sixteenth Annual International Conference of Computer Documentation* (ACM SIGDOC '98), Quebec City, Canada, 1998, pp. 127-136.

[28] R. Watkins, M. Neal, "Why and How of Requirements Tracing", *IEEE Software*, 1994 11(4), pp. 104-106.

[29] M. Yang, *COMIS – A Conceptual Model for Information Systems*, PhD thesis, NTH, Trondheim, Norway, 1993.