Nordisk kernesikkerhedsforskning
Norrænar kjarnöryggisrannsóknir
Pohjoismainen ydinturvallisuustutkimus
Nordisk kjernesikkerhetsforskning
Nordisk kärnsäkerhetsforskning
Nordic nuclear safety research

NKS-103
ISBN 87-7893-162-2

Traceability and Communication of Requirements in Digital I&C Systems Development. - Project Report 2004

Terje Sivertsen*, Rune Fredriksen*, Atoosa P-J Thunem*
Jan-Erik Holmberg**, Janne Valkonen**, Olli Ventä**
& Jan-Ove Andersson***

* Institute for Energy Technology, Halden, Norway
** VTT, Finland
*** Ringhals AB, Sweden

April 2005

Abstract

In 2004, the work has focused on providing a unified exposition on the issues studied and thereby facilitating a common approach to requirements handling, from their origins and through the different development phases. Emphasis has been put on the development of the TACO Traceability Model. The model supports understandability, communication and traceability by providing a common basis, in the form of a requirements change history, for different kinds of analysis and presentation of different requirements perspectives. Traceability is facilitated through the representation of requirements changes in terms of a change history tree built up by composition of instances of a number of change types, and by providing analysis on the basis of this representation. Much of the strength of the TACO Traceability Model is that it aims at forming the logic needed for formalising the activities related to change management and hence their further automation.

The work was presented at the second TACO Industrial Seminar, which took place in Helsinki on the 8th of December 2004.

Key words

Traceability, requirements, TACO, change management, digital I&C, systems development

NKS-103
ISBN 87-7893-162-2

Electronic report, April 2005

The report can be obtained from
NKS Secretariat
NKS-775
P.O. Box 49
DK - 4000 Roskilde, Denmark

Phone +45 4677 4045
Fax +45 4677 4046
www.nks.org
e-mail nks@nks.org

Foreword

This document constitutes the 2004 report for the project "Traceability and Communication of Requirements in Digital I&C Systems Development" (NKS-R project number NKS_R_2002_16). The report discusses the main issues covered in the project's research activities in 2004, and presents details with regard to the project organisation, activities, and further plans.

The purpose of the report is to document the continued research work and related activities within the TACO project, including the Second TACO Industrial Seminar (Helsinki, 8 December 2004). Particular emphasis has been put on providing a unified exposition on the issues studied and thereby facilitate a common approach to requirements handling, from their origins and through the different development phases. Together with the preproject report and the project report for 2003, the report provides an adequate basis for the final project report, to be finalized by the end of June 2005.

Halden, January 2005

Terje Sivertsen

Table of contents

Abbreviations

IFE	Institute for energy technology
NKS	Nordic nuclear safety research
NPP	Nuclear power plant
SKI	Swedish Nuclear Power Inspectorate
STUK	Radiation and Nuclear Safety Authority of Finland
TACO	Traceability and Communication of Requirements in Digital I&C Systems Development (NKS project number NKS_R_2002_16)
VTT	Technical Research Centre of Finland

Summary

The title of the reported project is "Traceability and Communication of Requirements in Digital I&C Systems Development", abbreviated TACO. The NKS project number is NKS_R_2002_16.

The overall objective of the TACO project is to improve the knowledge on principles and best practices related to the issues concretised in the preproject. On the basis of experiences in the Nordic countries, the project aims at identifying the best practices and most important criteria for ensuring effective communication in relation to requirements elicitation and analysis, understandability of requirements to all parties, and traceability of requirements through the different design phases. It is expected that the project will provide important input to the development of guidelines and establishment of recommended practices related to these activities.

The overall aim of the preproject, which was carried out in the second half of 2002, was to identify the main issues related to traceability and communication of requirements in digital I&C systems development. By focusing on the identification of main issues, the preproject provided a basis for prioritising further work, while at the same time providing some initial recommendations related to these issues. The establishment of a Nordic expert network within the subject was another important result of the preproject.

The project activities in 2003 constituted a natural continuation of the preproject, and focused on the technical issues concretised in the preproject report. The work concentrated on four central and related issues, viz.

- Representation of requirements origins
- Traceability techniques
- Configuration management and the traceability of requirements
- Identification and categorisation of system aspects and their models

The results from the preproject and the activities in 2003 were presented at the first TACO Industrial Seminar, which took place in Stockholm on the 12th of December 2003. The seminar was hosted by SKI.

In 2004, the work has focused on providing a unified exposition on the issues studied and thereby facilitating a common approach to requirements handling, from their origins and through the different development phases. Emphasis has been put on the development of the TACO Traceability Model. The model supports understandability, communication and traceability by providing a common basis, in the form of a requirements change history, for different kinds of analysis and presentation of different requirements perspectives. Traceability is facilitated through the representation of requirements changes in terms of a change history tree built up by composition of instances of a number of change types, and by providing analysis on the basis of this representation. Much of the strength of the TACO Traceability Model is that it aims at forming the logic needed for formalising the activities related to change management and hence their further automation.

The work was presented at the second TACO Industrial Seminar, which took place in Helsinki on the 8th of December 2004. The seminar was hosted by STUK.

1. Introduction

The title of the reported project is “Traceability and Communication of Requirements in Digital I&C Systems Development”, abbreviated TACO. The NKS project number is NKS_R_2002_16.

The present report documents the continued research work and related activities carried out within the TACO project in 2004. These activities have constituted a natural continuation of the activities in 2003, and have focused on providing a unified exposition on the issues studied and thereby facilitating a common approach to requirements handling, from their origins and through the different development phases. For this purpose, the project is developing a traceability-based, common approach to requirements handling (*the TACO Shell*). Underlying the approach is the *TACO Traceability Model*, based on the idea of a requirements change history built up by linking the different requirements together by way of a complementary set of basic requirements changes. On the basis of a representation of this change history, different kinds of analysis can be performed. The model facilitates traceability by representing requirements changes in terms of a change history tree built up by composition of instances of a number of change types, and by providing analysis on the basis of this representation. Much of the strength of the TACO Traceability Model is that it aims at forming the logic needed for formalising the activities related to change management and hence their further automation.

Chapter 2 briefly introduces the general problem of handling requirements of computer-based systems in nuclear power plants.

Chapter 3 presents the TACO common approach to requirements handling, called the *TACO Shell*. It is described how the shell utilizes the *TACO Traceability Model* as a basis for communication and understanding of requirements, and how different aspects of requirements can be handled within this model.

Chapter 4 presents some guidelines to the practical use of the TACO Shell in activities related to the different lifecycle phases.

Chapter 5 discusses the mathematical underpinnings of the TACO Traceability Model.

Chapter 6 contains the references in the report.

Appendix A gives an overview of the project organisation and activities.

Appendix B contains the minutes from the Second TACO Industrial Seminar, arranged in the premises of STUK, 8th of December 2004.

2. Computer-Based Systems in Nuclear Power Plants

The computer-based systems can appear nearly everywhere in a nuclear power plant. Computer-based systems are mainly associated with the I&C systems that have a large number of functions of different categories. Also the degree of complexity of the software and degree of inter-system communication needs and hardware solutions vary. A specific category is real-time applications.

The EUR (European Utility Requirements) document [European Utility Requirements for LWR nuclear power plants] defines the following levels of I&C systems:

0: process interface (sensors, transmitters, actuators, switchgear, etc.)
1: system automation
2: supervision & control
3: site management, off-site communication

The task of the I&C systems is to control the process by means of controlling process system equipment by gathering information from the process and process system equipment, by transmitting information between objects at the plant and by providing an interface to the operators of the plant. The I&C system includes the major interface means of communication between the power plant process and the surrounding world (Figure 1).

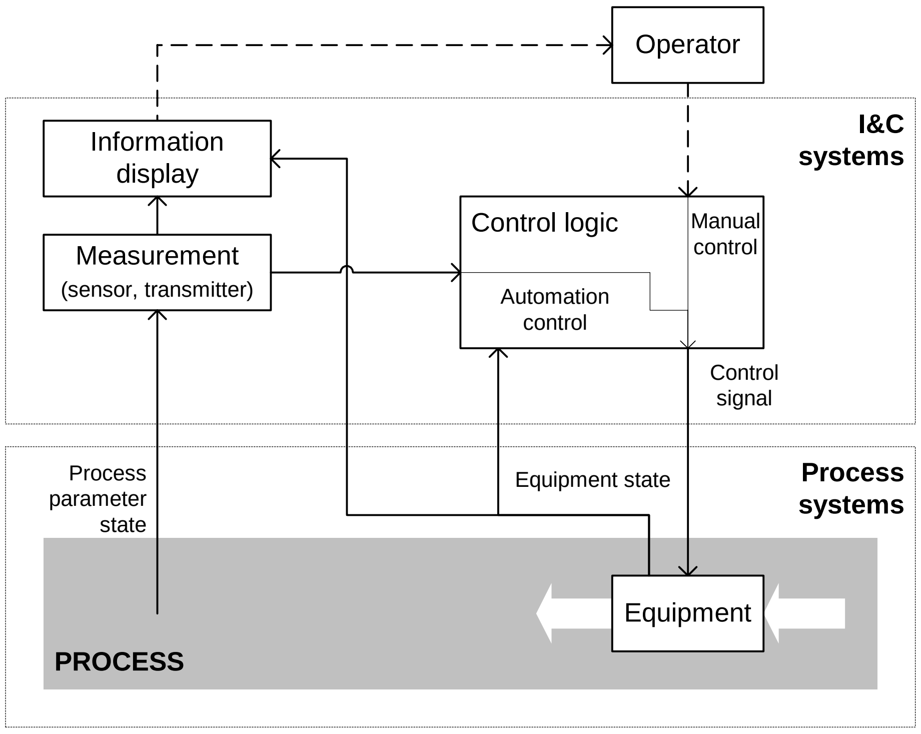

Figure 1. Interactions of the I&C systems, process systems and operators.

When defining requirements for an I&C system (i.e., a computer-based system), the functions attributed to the system are the basis for the requirements analysis. The key to successfully define the functional requirements is to define and analyse the functions at plant level. A plant level function can be e.g. reactor scram. This function involves several systems, both I&C systems, electrical systems, mechanical systems and fluid systems. The plant level functional analysis should result in definition of *interactions and associated requirements*. The next step is the functional analysis and function assignment in the system level leading to the *functional requirements* of the system. Thirdly *technical requirements* concerning the technical aspects of the system need to be defined.

To handle requirements of a computer-based system, like a smart sensor measuring certain safety-critical process parameter, a system model is needed to define and manage different types of requirements. In a model, the functionality of the system can be divided into the following modules [2]:

- system internal functionality
- human interaction functionality
- interface functionality
- safeguard functionality.

The idea is that such a model can be used as a framework for communicating requirements e.g. between plant safety engineers, automation engineers and software system designers.

3. The TACO Common Approach to Requirements Handling

The present chapter introduces the TACO common approach to requirements handling, called the *TACO Shell*. The idea is that the shell is a framework for traceability and communication of requirements, which can be filled with different contents to reflect the needs in different application areas. To facilitate its practical use, the TACO Shell is provided with guidelines, comprising *ingredients* and *recipes,* for filling and utilizing the TACO Shell. The TACO approach to requirements change management is based on a mathematically well-founded traceability model, called the *TACO Traceability Model*, where the introduction, changes, and relationships between different requirements, design steps, implementations, documentation, etc. are represented in terms of an extended change history tree. The traceability model adopted aims at forming the logic needed for formalising the activities related to change management and hence their further automation. By complementing the model with appropriate terminology, data structures and guidelines for use, the model can be adapted to the different needs related to management of changes in computer-based systems, including safety-critical and security-critical systems.

3.1 Traceability - a Basis for Communication and Understanding of Requirements

The three main aspects in the objectives of the TACO project are

- effective communication in relation to requirements elicitation and analysis;
- understandability of requirements to all parties; and
- traceability of requirements through the different design phases.

The approach followed in the TACO project is to utilize traceability of the requirements, through an appropriate traceability model, to facilitate communication and understandability:

Communication: Effective communication in relation to requirements elicitation and analysis relates to the common understanding of the requirements, which again requires adequate communication. The TACO Traceability Model supports communication by providing a common basis, in the form of a requirement change history, for different kinds of analysis and presentation of different perspectives. By way of example, the model can be used for generating subsets of the change history showing the backwards or forwards traceability of given requirements. The TACO approach provides guidelines for how to utilize these possibilities in practical work.

Understandability: In almost all cases within the software industry, requirements understanding continues to evolve as design and development proceeds. For many large systems, the requirements are never perfectly understood or perfectly specified. By relating the development of the requirements through the TACO Traceability Model, the different agents involved in the development can at any time relate a given version of a requirement to its origins, its different development stages, and its relationship to other requirements. Understandability can be further improved by providing traceability between different perspectives and forms of a requirement. By way of example, the understandability of a formal expression of a requirement, or a perspective intended for particular expertise, can be improved by relating the relevant paragraph to another paragraph stating the requirement in a different way.

It is implied by the foregoing discussion that traceability plays a central role in the discipline of requirements engineering, which aims at developing standard and systematic ways to elicit, document, classify, and analyse requirements. Since these are among the most critical activities in the software engineering process, the effectiveness of traceability management may have a significant effect on the quality of this process, and thereby on the success of the application. Of particular importance to this success is the communication and understandability of the requirements to be met by the system, and how these well-understood and well-documented requirements can be traced throughout the different development phases.

Management of changes is closely related to the maintainability of a software system. Typically, the requirements for a given system undergo many changes before the development is completed. These changes may be due to changes in the prospected operation environment, but may also happen simply as a result of improved insight during the development. The task of managing alteration of the requirements is closely related to requirements traceability. In fact, work on requirements traceability can to a certain extent be seen as a response to the need for keeping track of these changes. One benefit of traceability is the localization of the side effects of a modification and the identification of relationships that must be reconfirmed, thereby increasing the assurance that when changes are necessary they will be complete and consistent.

3.2 The TACO Shell

The *TACO Shell* is the overall TACO framework for requirements handling, and represents a generic approach to lifecycle-oriented, traceability-based requirements management. The TACO Shell comprises the overall methodology, the *TACO Traceability Model*, and the different guidelines related to its contents (*ingredients*) and use (*recipes*). By varying the ingredients and recipes, the shell can be used for the development of different kinds of target systems, with different requirements origins, different emphasis on quality attributes, and different selection of dependability factors.

3.3 The TACO Traceability Model

The TACO Traceability Model adopts several of the ideas to fine-grained traceability presented in [4]. Accordingly, traceability is facilitated by representing the requirements changes in terms of a change history tree built up by composition of instances of seven different change types, and to provide analysis on the basis of this representation. The change types correspond to the following generic actions performed on requirements, or more generally, *paragraphs* (from [4]):

- *Creating* a new paragraph with no prior history.
- *Deleting* an existing paragraph.
- *Splitting* an existing paragraph, thereby creating a number of new paragraphs.
- *Combining* existing paragraphs by a new paragraph.
- *Replacing* existing paragraphs by a new paragraph.
- *Deriving* a new paragraph from existing paragraphs.
- *Modifying* a paragraph without changing it's meaning.

The change history can be represented by a tree where the paragraphs constitute the nodes. The tree representation constitutes an appropriate basis for different kinds of analysis, including finding (see chapter 5)

- all *initial* paragraphs;
- all *deleted* paragraphs;
- all *applicable* paragraphs;
- the complete *history* of a paragraph;
- the complete *backwards traceability* from a set of paragraphs;
- the complete *forwards traceability* from a set of paragraphs;
- the *legality* of a proposed requirements change.

The possibility to find the backwards or forwards traceability from a set of requirements facilitates backwards and forwards branch isolation and analysis of the change history. The versatility of the representation can be further improved by extending the representation of the paragraphs to include different parameters that classify the requirements, provide additional information, etc. Possible parameters are discussed later in the report.

When it comes to the representation of the actual parameters, it is important to distinguish between (1) the information that is essential to identify the paragraph, and (2) the various information associated to this parameter. Conceptually, and from a perspective of modularity, it is useful to let the nodes in the change history tree represent the necessary and sufficient information related to the identity of a paragraph. In the TACO Traceability Model, a paragraph is represented by the combination of a unique identifier for this paragraph and a version number to distinguish several versions of the same paragraph. At any time, only the latest version of a paragraph can be an applicable paragraph. That is, a new version of a paragraph is introduced only if this replaces old versions. In any case, it is possible to make duplicates of a paragraph when these are treated as different paragraphs. This can also be used for representing different variants of the same requirement, possibly with "application conditions" attached as guidelines to every single variant. Each variant will however be represented with a separate paragraph.

It is important to note that concepts similar to those described above for the TACO Traceability Model can be found in commercial tools for version control and configuration management. Although the change types might have other names, they typically resemble those defined here. In general, however, these tools do not offer an identifiable, formally defined traceability model, and leave to the user to define the actual semantics underlying the different change types. The strength of the TACO Traceability Model is that it aims at forming the logic needed for formalising the activities related to change management and hence their further automation.

Conceptually, we can think of a node of the change history tree as a versioned paragraph, represented by a pair of a paragraph identifier and a version number. In the following we will use the change history in Figure 2 as an example.

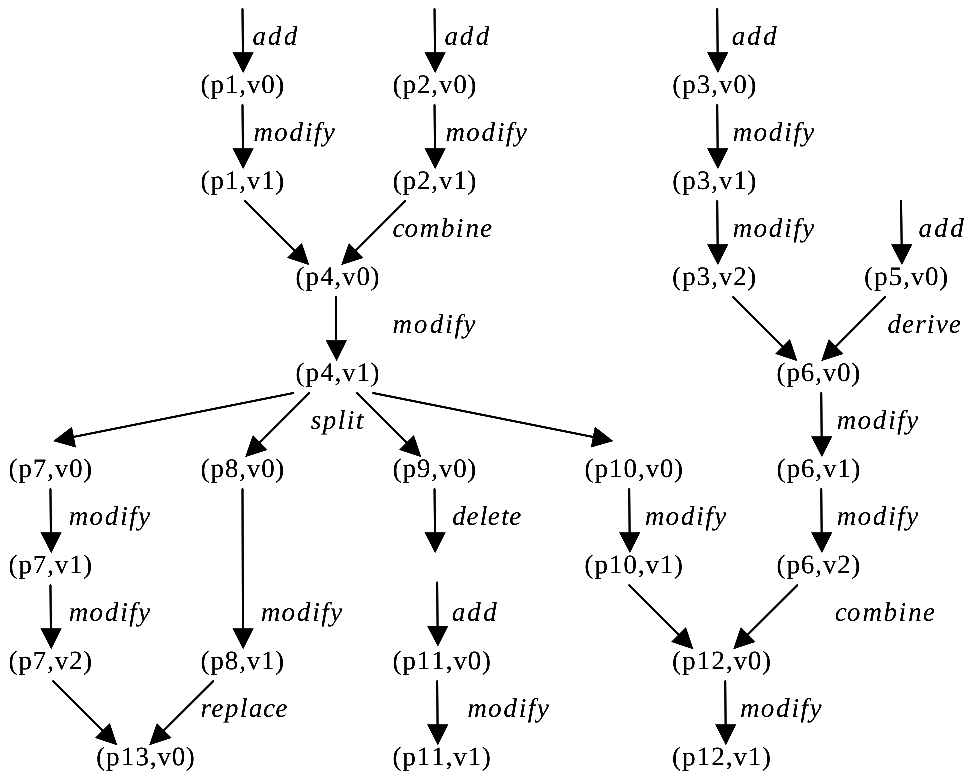

Figure 2. The example change history.

The development of the requirements in Figure 2 starts with the introduction of the paragraphs p1, p2, and p3. At later stages, another two new paragraphs are introduced, viz. p5 and p11. All the other paragraphs are developed on basis of these five paragraphs. Paragraphs p1 and p2 are first modified and then combined into a new paragraph p4. After a modification, this paragraph is split into four separate paragraphs p7 to p10. The latter of these paragraphs is modified and then combined with p6, originally derived from paragraphs p3 and p5, giving paragraph p12. Note that, at any point in the development of the paragraphs, at most one version of a paragraph is applicable (in the sense that it is the valid version of the paragraph).

It is certainly possible to represent the change history tree textually in such a way that the temporal relationships between the different changes are maintained. How this can be done is discussed in chapter 5.

Let us now consider the other information attached to a paragraph. As has been argued in the foregoing, it is not necessary to represent this information in the change history tree. The purpose of the tree is to give a complete representation of the changes and how they are related to each other. What about the other information, including the actual text of the paragraph? Formally, we can think of these relations in terms of some basic mathematical concepts:

- *Sets*: These are finite collections of objects of some type, and can be used for representing subsets of the paragraphs. By way of example, the classification of paragraphs with respect to Business plan, Requirements document, Design specification, etc, can be represented by means of separate, maybe overlapping sets corresponding to the different classification terms. Finding, say, all Business plan related requirements is then trivial, since they are given by the corresponding set. Checking whether a requirement belongs to the Business plan is also easy and can be done simply by checking whether the given paragraph is a member of the corresponding set. On the other hand, finding the class of a given paragraph cannot be done by simple look-up but involves checking all the different sets for membership.
- *Mappings*: These are functions from a source set to a target set, and can be used for assigning information to the paragraphs in a simple look-up fashion. With this solution, e.g. the classification of paragraphs can be represented by mappings from the paragraphs to their classification. Finding the classification of a requirement is then simple, since it reduces to looking up the classification of that requirement. Finding all requirements is possible, but less trivial than for sets, as it involves selecting all requirements that are mapped to a certain term. On the other hand, the concept of relation is more convenient if there may be more than one class for a requirement.
- *Relations*: These are more general than mappings, since they allow an element in the source set to be associated to more than one element in the target set. With this solution, finding the classification of a requirement involves finding all elements in the target set (the classes) that are related to the given requirement. Finding all requirements related to a certain class can alternatively be understood as the inverse relation.

Sets can be considered as being implemented as simple lists. Mappings and relations can be considered as being implemented as tables. As we will see in the continued discussion, these representation concepts will suffice for representing all information associated to the requirements. It is of course possible to represent the same information in other ways as well, as long as consistency is maintained.

A basic piece of information related to a requirement is certainly the statement (phrasing) of the requirement. Assuming that (at most) one statement is associated to each requirement, we may think of this information as being available by means of a mapping from versioned requirements to their statements, see Figure 3.

Requirement	**Statement**
(p1,v0)	<Statement of version v0 of paragraph p1>
(p1,v1)	<Statement of version v1 of paragraph p1>
(p2,v0)	<Statement of version v0 of paragraph p2>
...	...
(p13,v0)	<Statement of version v0 of paragraph p13>

Figure 3. Mapping from requirements to their statements.

As is evident from Figure 3, the statement of a given requirement can be found by simple look-up of in the table implementing the mapping. The table can be utilized in different ways. By way of example, finding all relevant requirements can be found by filtering the mapping with respect to the applicable paragraphs to find the subset of the mapping that relates to applicable paragraphs only. Filling in the relevant information is an obvious task of an information system designed to support the use of the model.

Other useful information can be represented in the same way. By way of example, a recurrent problem with modernization projects is the difficulties of recapturing both the “what” and the “why” of a requirement. In the TACO Traceability Model, the “what” is covered by the table in Figure 3. In a similar way, the “why” of the requirements can be covered by a similar mapping from requirements to comments giving information on the background, motivation, reasons, etc. for including the requirements.

A possible utilization of the TACO Traceability Model is in the identification of relative influences, correlations, and conflicts between safety/security countermeasures and other dependability factors. On this basis, guidelines to the use, implementation, and verification of the different change types can be developed. These guidelines would have to reflect the identified relative influences, correlations, and conflicts in the sense that they provide a better basis for controlling the effects of changes. The guidelines should include descriptions on how different techniques can be applied for this purpose, such as the use of formal specification and proof for demonstrating the correct derivation of requirements, coding standards for implementation of specific design features, etc.

3.4 Traceability Techniques

The traceability technique used by the TACO Traceability model has a structure of a *tree*. In the following, we will briefly look into two widely applied traceability techniques and compare their features with those of the change history tree proposed.

Traceability tables: In a traceability table, the requirements are listed along the horizontal and vertical axes, and relationships between requirements are marked in the table cells, see Figure 4.

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10	p11	p12	p13
p1													
p2													
p3													
p4	*	*											
p5													
p6			*		*								
p7				*									
p8				*									
p9				*									
p10				*									
p11													
p12						*				*			
p13							*	*					

Figure 4. A traceability table for the example change history.

The information in the traceability table can easily be found from the change tree, and constitutes a subset of the arcs in this tree. Comparing the basic traceability table representation with the change tree representation, we note the following:

- The tree gives a visual, graphical view of how the different requirements are related to each other, and may therefore be more effective when it comes to communication and understanding.
- While the size of the tree grows linearly with the number of requirements, the size of the table grows with the square of the number of requirements. A change history tree with 100 versioned requirements could fit into a page or two, whereas the corresponding traceability table would contain 10.000 entries, of which perhaps 1-2% would be used. The way this is handled in practice depends on how the tables are implemented.
- In contrast to the change history tree, the table does not state explicitly that a requirement with an empty row and column means that the requirement has indeed been created. One solution is to assume that all the requirements listed have in fact been introduced. As such, all requirements with an empty row and column must be counted as applicable requirements. Alternatively, the table must be appropriately extended with the necessary information.
- In contrast to the change history tree, the table does not indicate whether a requirement has been deleted. Other requirements may still depend on the requirement, but the requirement may later loose its importance and therefore be removed from the set of applicable requirements. Again, there is a need to extend the basic table with the necessary information.
- In contrast to the change tree, the simple type of traceability table shown in Figure 4 does not distinguish between the different ways in which requirements depend on each other. Including this information in the column solves this for the changes represented. In addition, we need to represent information related to the creation or deletion of a paragraph.
- Even if it in principle were possible to design a traceability table that represents the different relationships and changes in the change tree, the table would still miss the conciseness, clarity and usefulness of the change tree with respect to its basis for communication and understandability. This includes the temporal nature of the requirements changes, i.e. the fact that they can be partially ordered with respect to the sequence in which they were introduced. The information can be derived from the table, but is not visualized in a simple way as in the change tree.

Traceability lists:

A traceability list is a simplified, compact form of a traceability table, where the interdependencies are represented by listing the depends-on relation for each requirement. A traceability list corresponding to the change tree example is given Figure 5.

Requirement	**Depends-on**
p4	p1, p2
p6	p3, p5
p7	p4
p8	p4
p9	p4
p10	p4
p12	p6, p10
p13	p7, p8

Figure 5. A traceability list for the example change history.

Comparing the basic traceability list representation with the change tree and the basic traceability table representation, we note the following:

- Similar to the table, the list lacks the enhanced visual, graphical view provided by the tree.
- Similar to the tree, the size of the list grows linearly with the number of requirements.
- Similarly to the table, the list in Figure 5 does not distinguish between different versions of the requirements. Such an extension would be possible by treating each versioned paragraph separately.
- The list does not state explicitly that a requirement has been created. Instead one is forced to assume that all the requirements in the right column in fact been introduced, but this does not cover the requirements upon which no other requirement depends. One solution is to include all requirements in the table, and not only those depending on other requirements. All requirements with an empty row would then be counted as applicable requirements.
- Similarly to the table, the list does not indicate whether a requirement has been deleted.
- Similarly to the table, the simple list in Figure 5 does not distinguish between different ways in which requirements depend on each other.
- Similarly to the table, the list lacks the clarity and usefulness of the change tree with respect to its basis for communication and understandability.
- The list is more difficult to use than the table when it comes to finding the inverse of a relation. Finding all requirements that depend on a certain requirement involves searching through the whole table. This is not adequate when it comes to using the list as a medium for communication and understandability.

3.5 Representation of Requirements Origins

As has been indicated earlier in the report, the TACO Traceability Model can incorporate different information by means of parameters or data structures of different kinds. An important piece of information is the origin of a requirement. By way of example, some requirements may originate from technical requirements to the system, some from standards adopted, some from the business plans, etc. Since a requirement may have more than one origin, it is inadequate to represent requirements origins by means of a simple mapping from requirements to origins. Instead we will use the concept of a binary relation. Typically, the domain of this relation will first of all consist of the paragraphs introduced with no prior history (in the change history tree), but we should also open up the possibility to relate paragraphs with a prior history to be related to a requirement origin, e.g. when a paragraph is modified to fit a certain origin.

With reference to the change history tree example, we can think of the requirements origins as being represented by some relation like the one indicated in Figure 6.

Requirement	**Origin**
(p1,v0)	Technical requirement <with reference>
(p2,v0)	Technical requirement <with reference>
(p3,v0)	Guideline <with reference>
(p3,v0)	Technical requirement <with reference>
(p3,v2)	Standard <with reference>
...	...

Figure 6. Relation between requirements and origins.

The relation in Figure 6 indicates that versions v0 of requirements p1 and p2 originate from certain technical requirements (with precise references to these, such as document, section, paragraph, etc.), version v0 of paragraph p3 originates both from a guideline and a technical requirement, version v2 of paragraph p3 originates from a standard, for example as a result of reformulating the previous version to fit the standard, etc.

With the representation of requirements origins, the origin of a requirement can be included in different kinds of analysis of the requirement's lifecycle:

- *Forwards traceability* can be analysed on the basis of a given requirements origin to see how the requirements of the given origin are handled throughout the system life cycle.
- *Backwards traceability* can be analysed on the basis of a particular version of a given requirement, somewhere in the system life cycle, to get a complete picture of the origins of this requirement.

The traceability model can be complemented by data structures providing an adequate representation of the requirements origins and using these in combination with the model to perform different kinds of analysis. By way of example, the different paragraphs in a given standard can be represented by a mapping from paragraph identifiers to paragraph statements. If the identifiers used for the requirements origins have the same form and correspond to those used in the latter mapping, the two data structures can be used in combination to find the actual statement in the paragraph constituting the origin of the given requirement. Another useful data structure is the set of applicable paragraphs in a given standard as a subset of these paragraph identifiers. It is then straightforward to find the requirements in the traceability model that address the relevant parts of the standard. This can be utilized to demonstrate that all the relevant parts of the standard have been covered by the requirements. By analysing the forward traceability of these requirements, one can use this to demonstrate that the paragraphs in the standard have been handled appropriately during the development of the system.

Analogously to the representation of the requirements origins, the concept of binary relations can be used to represent different classifications of the requirements. By way of example, technical requirements for functions, systems, equipment, and associated design and life cycle processes, can be divided into four groups:

- Requirements that apply to the functions concerning the assurance of functionality, performance and reliability.
- Requirements that apply to the design characteristics such as redundancy, diversity, testability and separation.
- Requirements concerning the equipment features for the assurance of seismic and environmental durability and electro-magnetic compatibility.
- Requirements associated with the quality assurance, quality control, verification and maintenance.

The use of a binary relation to represent the classification of technical requirements facilitates overlapping classifications in the sense that the same requirement can be classified in more than one way. As before, the representation of the relationship can be utilized in different kinds of analysis. By way of example, the requirements belonging to a certain class can be found by filtering the relation against this class. Similar representations can be made to relate the requirements to different purposes, parts of the systems, modules, etc. These extensions of the information and knowledge related to the requirements can be done in a flexible manner with the use of the basic representational structures introduced, i.e. sets, mappings, and relations, and can be done in a way and to an extent that suits the particular system at hand.

From a knowledge management point of view, the core of the requirements management process is the requirements database where the requirements are stored in a structured way. There are several ways to build the structure, representing three origins or points of views to the requirements database, viz. user oriented, objective oriented, and plant model oriented perspectives. Again, this information can be represented by means of a binary relation, which again can be used to find the viewpoint of a requirement, or all the requirements related to a given viewpoint.

In general, system requirements are goals the system must supply or qualities it must possess in order to fit its intended use. In nuclear safety, the emphasis is on deriving the safety technical requirements traceable from the safety case documentation, standards, etc. The requirements for design and operation of nuclear power plants can be derived from the basic safety principles that include safety objectives, management principles, defence in depth principles and technical principles. These principles are reflected in the national and international rules, standards and guidelines. By identifying and representing the origins of the safety technical requirements as described in this report, their derivation from these origins can be followed throughout the system lifecycle. How the different requirements actually are derived from the origins can be made subject to review by checking the backwards traceability towards their origins and how the correctness of the different steps is demonstrated. For this purpose, the idea of a fine-grained traceability model in terms of individual requirements changes appear to provide the detail necessary for this kind of review, analysis and demonstration. The model also gives an appropriate framework for formal demonstration of the correctness of each requirement change (or design step), where this is relevant.

3.6 Configuration Management

Requirements traceability and change management are a part of configuration management and should therefore be treated as such. This means that they must follow a common approach. Typically, the requirements for a given system undergo many changes before the development is completed. These changes may be due to changes in the prospected operation environment, but may also happen simply as a result of improved insight during the development. As implied above, the task of managing changing requirements is closely related to requirements traceability. In fact, work on requirements traceability can to a certain extent be seen as a response to the need for keeping track of these changes. As is stated by [5], one benefit of traceability is the localization of the side effects of a modification and the identification of relationships that must be reconfirmed, thereby increasing the assurance that when changes are necessary they will be complete and consistent.

In the TACO project, the configuration of a system is viewed as a collection of specific versions of the configuration items combined according to specific build procedures to accomplish a particular purpose. In accordance to this view, the purpose of configuration management is to identify the different configurations of a system in order to facilitate change management and the maintenance of integrity and traceability throughout the system life cycle. In relation to software development, project documents as well as code can be put under configuration control. Furthermore, the software process should be requirements-centred (or user-centred). In particular, software configuration management should support traceability from the initial or negotiated requirements to the finally delivered code.

The TACO Traceability Model fits well with this view on software configuration management, in particular on the management of requirements changes. For each version of a product, work product, or even (sub-) process delivered or employed in the system development, it is in principle possible to associate a set of versioned requirements. That is, the configuration of some (sub-) product or (sub-) process includes the versioned requirements associated to it. Note that the versions of these requirements may later be replaced or modified.

By way of illustration, suppose that the example of the change history constitutes the requirements associated to a certain work-product, e.g. a design document. That is, the design document is based on, and should satisfy, the applicable requirements. These requirements (in their most recent versions) therefore constitute part of the configuration of the design document. Certainly, the design document may itself exist in several versions, with their separate configuration. In this particular example, it is natural to think of the configurations of these earlier versions of the design document as associated to subtrees of the change history tree. The subtree associated to an earlier version of the design document can easily be identified by way of backwards traceability from the set of versioned requirements included in the configuration of this document.

With reference to the formal framework introduced earlier in this chapter, the relationships between products/processes and versioned requirements can be represented by means of mappings. This is exemplified in Figure 7.

Work product	**Requirements**
Design doc. 1, version 1	(p3,v2), (p5,v0), (p7,v2), (p8,v1), (p12,v1)
Design doc. 1, version 2	(p3,v2), (p5,v0), (p11,v1), (p12,v1), (p13,v0)
...	...

Figure 7. Mapping from work products to requirements.

By inspecting the change tree, it is easy to see that the first version of the document in Figure 7 reflects the requirements before p7 and p8 were replaced by p13, and p11 introduced. The subtree can easily be generated from the information provided by the mapping.

Ideally the requirements management is a process where requirements are elicited, analysed and specified from general ones towards implementation specific descriptions. During the life cycle of the plant (in case of new nuclear power plants) or the system (in case of I&C upgrading projects), various documents are extracted from the collection of requirements such as overall and detailed requirements specification, safety and reliability assessments, and plans for designated activities. All of these documents, and the collection of requirements from which they are extracted, can be treated as exemplified above, i.e. in terms of mappings from work documents to versioned requirements. Alternatively, or in combination with this, the different versions of these work products can themselves be included in the change history tree. With these options, we have at least the following three alternative ways of treating their configurations, all of which provide the same information:

- The work products are not included in the change history tree but associated to the requirements by means of mappings. This corresponds to the example in Figure 7.
- The work products are included in the change history and associated to the requirements by means of mappings. The introduction of a work product is represented in the change tree by the creation of a new paragraph with no prior history.
- The work products are included in the change history and associated to the requirements by means of the derivation of new paragraphs from existing paragraphs. In this way, requirements and work products are treated in the same way, except that additional information can be added to relate the paragraphs to the types of work product they represent. The mapping from work products to requirements is optional since the information it represents can be found from the change history tree.

Certainly, much of the foregoing can be approached with existing configuration management tools. What the TACO Traceability Model offers is first of all the logic needed for formalising the activities related to change management and hence their further automation.

4. TACO Guidelines

The TACO project aims at providing input to the development of guidelines and establishment of recommended practices related to requirements elicitation and analysis, understandability of requirements to all parties, and traceability of requirements through the different design phases. In this chapter, guidelines will be presented to the practical use of the TACO Shell in activities related to the different lifecycle phases. The guidelines can be seen as comprising *ingredients* and *recipes* for filling and utilizing the TACO Shell. By gradually complementing the TACO Shell and the TACO Traceability Model with appropriate terminology, data structures and guidelines for use, the model can be adapted to the different needs related to management of changes in computer-based systems, including safety-critical and security-critical systems. By way of example, the model can organize communication and analysis of requirements by generating subsets of the change history showing the backwards or forwards traceability of given requirements. The TACO guidelines help to utilize these possibilities in practical work.

By varying the ingredients and recipes, the TACO Shell can be used for the development of different kinds of target systems, with different requirements origins, different emphasis on quality attributes, and different selection of dependability factors. The TACO guidelines can be developed on a continual basis to fit the use, implementation, and verification of the different change types. The guidelines should include descriptions on how different techniques can be applied, such as the use of formal specification and proof for demonstrating the correct derivation of requirements, coding standards for implementation of specific design features, etc.

4.1 Validity of Requirements Changes

Software development needs to deal with changes to the requirements, also after the requirements specification phase ideally is completed and the requirements frozen. The evolutionary nature of software implies that changes will have to be anticipated. Other changes may be necessary due to our evolving understanding about the application under development.

One of the lessons learned in the software engineering area is that software should be designed for change. The focus in the present report is on how to manage the evolution of the requirements in this situation. The present section deals with how the TACO Traceability Model can be utilized in the validation of the changes representing this evolution.

The TACO Traceability Model is based on a number of change types that can be employed to manage requirements changes throughout the life cycle of a system. Each change introduced in the life cycle should in principle be validated. Depending on the level of rigidity or formality employed, the validity of a change can be done in a variety of ways, from a simple inspection to a formal mathematical proof. Notwithstanding these differences, we will in the following concentrate on what validity in general means for the different change types. Validity should not be confused with the legality of changes, discussed in section 5.9. While validation concerns the semantics of the changes, the legality of a change can be checked mechanically from the structure of the change history tree.

Creating: Applied on requirements, creating a new paragraph with no prior history involves introducing a new requirement. The validity of the requirement involves both its *correctness* with respect to its intended meaning, its *completeness* with respect to its coverage of its intended meaning, and its *consistency* with other requirements. In short, the validity of a new requirement requires that it faithfully reflects the intended meaning and that it is not in conflict with other requirements. This is the only change type that is allowed to introduce new requirements or new aspects of requirements that are not already covered by existing paragraphs.

Deleting: Deleting an existing paragraph involves that a requirement in fact is withdrawn from the set of requirements. A requirement can be deleted if either the requirement in itself is no longer valid, or it is covered by other requirements. To demonstrate the validity of the change therefore either involves showing that it is the intention to withdraw the requirement as such or showing that it can be derived from other requirements.

Splitting: Splitting an existing paragraph involves creating a number of new paragraphs that collectively replaces the given one. Applied on requirements, a paragraph split is valid only if the requirements given in the new paragraphs together cover the replaced requirement, but not more. In other words, splitting a paragraph is not valid if the new paragraphs require more or less than the replaced paragraph.

Combining: Combining a set of existing paragraphs involves creating a new paragraph on basis of the existing ones, without deleting any of the existing paragraphs. Applied on requirements, a combination of paragraphs is valid only if the new paragraph covers the given paragraphs, but not more. In other words, combining a set of paragraphs is not valid if the new paragraph requires more or less than the given paragraphs.

Replacing: Replacing a set of existing paragraphs involves creating a new paragraph that replaces the existing ones. The validity criterion is identical to that of combination. Replacing a set of paragraphs and splitting an existing paragraph are inverse changes.

Deriving: Deriving a new paragraph from a set of existing paragraphs involves creating a new paragraph on the basis of the existing ones, without deleting any of the existing paragraphs. Applied on requirements, deriving a new paragraph is valid only if the requirement is *one* of the possible results/consequences of the requirements it is derived from.

Modifying: Modifying a paragraph should involve no changes to its meaning. The new requirement should therefore cover the replaced requirement, but not more.

Attempts on demonstrating the validity of individual changes may reveal flaws in the requirements management, such as introducing new paragraphs in a paragraph split that actually adds new requirements that are not covered by the replaced requirement. Detecting such flaws can be utilized in the requirements change process to produce an appropriate requirements change history, such as specifying such added requirements in terms of separate changes of type *creating new paragraphs with no prior history*. Similarly, insufficient coverage of the replaced requirement in a split change can be made "clean" by complementing the split with separate changes of type *deleting an existing paragraph*. In this way, an invalid change can be replaced by a set of valid changes, and the need for demonstrating the validity of the different changes can be made explicit.

4.2 Formal Review and Test of Requirements

Due to the high costs associated with defects slipping through the requirements specification phase, formal review and test of the requirements documents are usually highly prioritised activities. Industrial experience shows that very often a significant fraction of the most critical software defects are introduced already in the requirements specification. Of this reason, it is generally recommended to carry out tests on this specification that are as near as exhaustive as possible, and for this purpose, the use of a formal approach is often advocated.

Requirements analysis and requirements validation have much in common, but the latter type of activity is more concerned with checking a final draft of the requirements document which includes all system requirements and where known incompleteness and inconsistency has been removed, see [3]. As such, it should be planned and scheduled in the quality plan for the project, and be carried out in accordance with good quality assurance practice.

One of the theses behind the present report is that the TACO Traceability Model can be used for revealing and correcting several kinds of shortcomings discovered during the validation of the requirements document. This is true in particular for problems related to lack of conformance with the standards employed. The validation of the requirements against a given standards can be carried out by utilizing the information included about the origins of the requirements, as was described in section 3.5. Such a validation could include the following steps:

1. Add all the requirements from the given standard by creating new paragraphs. If certain requirements are found irrelevant, the exclusion of these can be made explicit by deleting these paragraphs. This also makes explicit the need to validate their exclusion.

2. Check that the applicable and deleted paragraphs together constitute the complete set of requirements given in the standard. This can partly be automated by keeping these requirements on file.
3. Validate the change history related to the applicable paragraphs originating from the standard, utilizing the guidelines listed in section 4.1.
4. Validate the deletion of paragraphs originating from the standard, utilizing the guidelines listed in section 4.1.

Using the TACO Traceability Model in validating the requirements document may be done in the context of a formal requirements review meetings, in accordance with general guidelines to such meetings. Requirements validation may also take other forms, like prototyping, model validation, and requirements testing, but the focus in the present report is on the utilization of the requirements change history in the review meetings. For further reading on formal review meetings, see [3].

Requirements reviews are conventionally carried out as a formal meeting involving a group representing the stakeholders. The general idea is that the system stakeholders, requirements engineers and system designers together check the requirements to verify that they adequately describe the system to be implemented. Traceability and requirements changes are of course only part of the concern at such a meeting. The TACO Traceability Model may however provide important assistance for discovering requirements problems related to requirements conflicts or lack of conformance to standards and other requirements origins.

In the end, the requirements traceability is itself a concern of the requirements review. As discussed in [3], the requirements should be unambiguously identified, include links to related requirements and to the reasons why these requirements have been included. Furthermore, there should be a clear link between software requirements and more general systems engineering requirements. This relates to the obvious fact that the software engineering activity is part of the much larger systems development process in which the requirements of the software are balanced against the requirements of other parts of the system being developed [1]. Furthermore, the software requirements are usually developed from the more general system requirements, and thus the traceability and consistency with these requirements is a basic premise for a successful process and its resulting product.

4.3 Correctness of Implementation

The correctness of implementation is a quality that characterizes the ability of the application to perform its function as expected [1]. Reasoning about correctness therefore requires the availability of the functional requirements, and we say that the application is functionally correct if it behaves according to the specification of these requirements.

In principle, correctness is in this context a mathematical property that establishes the equivalence between the software and its specification. In practice, the assessment of correctness is done in a more or less systematic manner, depending on how rigorously the requirements are specified and the software developed. In any case, the assessment requires that the requirements can be traced forward to their implementation, and vice versa.

The TACO Traceability Model supports the assessment of correctness by relating the requirements and their implementation through the change history tree. This relationship can be utilized in both a forwards and backwards fashion. The TACO shell provides both forwards and backwards traceability analysis, without requiring separate links for forwards and backwards traceability. The different types of analysis can be defined on the basis of one and the same representation of the change history tree.

In general, a forward traceability approach to assessment of correctness would take the specified requirements as starting point, and then demonstrate that all the requirements have been correctly implemented. Analogously, a backward traceability approach would take the implementation as starting point and check the consistency with the requirements. Of these two, the forward approach probably fits better with respect to a conventional approach to correctness assessment.

In practice, using the TACO Traceability Model for assessment of functional correctness can be done in terms of the following steps.

1. For each requirement introduced, indicate whether it is a functional requirement. This can be done by means of mappings, as described in section 3.3.
2. For each implementation of a requirement, indicate - by means of mappings - that it is an implementation.
3. For each functional requirement introduced, check that the forward traceability leads up to an implementation of this requirement. This can be done by:
4. For each functional requirement, check that the requirement is correctly implemented by validating the sequence of changes leading from the requirement to its implementation.

4.4 Requirements Understanding

One important aspect of the requirements understandability relates to the understanding of the interface between the application to be developed and its external environment (such as the physical plant). This requires that the software engineers understand the application domain and communicate well with the different stakeholders. To facilitate this communication, it might be necessary to specify the requirements in accordance with the different viewpoints the stakeholders have to the system, where each viewpoint provides a partial view of what the system is expected to provide. As a consequence, the requirements specification will cover different views on the same system, giving an additional dimension to the question of consistency between the different requirements. An important task of the software engineers is to integrate and reconcile the different views in such a way that contradictions are revealed and corrected.

In order to cope with the complexity of the resulting set of requirements, it is advisable to classify and document the requirements in accordance with the views they represent. This way of separating the concerns can provide a horizontal, modular structure to the requirements. Modularity provides several benefits in the requirements engineering process, including the capability to understand the system in terms of its pieces. This first of all relates to the fact that modularity allows separation of concerns, both with respect to the different views represented by the different stakeholders' expectations to the system and to different levels of abstraction. This makes it easier for the different stakeholders to verify their requirements, while at the same time providing a means for handling the complexity of the full set of requirements. The TACO Traceability Model can be adopted to facilitate this separation of concerns by relating requirements to the views they reflect. This can be utilized in different kinds of analysis of the requirements throughout the development of the system.

Some of the stakeholders may be unable to read the types of specifications preferred by the software engineers or mandated for the application. In such cases, the needs of the different stakeholders can be reconciled by providing (horizontal) traceability links between the, possibly formal, specifications used by the software engineers and more informal, natural language based expression of the same requirements. One could even consider providing links between the requirements and the user manual within the same traceability model. This could be utilized both for communication purposes and for the purpose of developing the user manual in parallel to the engineering of the requirements, which in some cases may be a recommended practice.

5. Mathematical Underpinnings

This chapter discusses the mathematical underpinnings of the TACO Traceability Model in terms of a functional specification of the change history tree, the different change types, and different kinds of analysis that can be performed on basis of this representation. The specification is written in the algebraic specification language HALDEN ASL, supported by the HALDEN Prover [6][7][8][9]. (Introduction to the use of HALDEN ASL and the HALDEN Prover is outside the scope of this report). The resulting specification can be used directly in the implementation of a simple, generic configuration management system, providing part of tool support for the TACO Shell.

The Traceability Model will be specified in two layers, reflected in a hierarchy of two specifications. In the "lower" specification, the different change types will be specified inductively as generators, thus providing a data structure for the change history. At the top of this specification, the different change types will be specified as operators, checking that the given change is legal and producing a lower layer representation together with the set of applicable paragraphs. By applying these operators, only legal change histories will be represented.

A paragraph will be specified as a pair consisting of a paragraph label and version label:

```
PAIR(PAR,VERSION)
```

The label types are left unspecified as two formal type parameters `PAR` and `VERSION`. Pragmatically, the two parameters are treated as so called *primitive data types*. For the purpose of specifying and verifying the change management, it is not necessary to specify these data types any further. Intuitively, that is why they are called primitive data types.

For a given configuration, we require that each paragraph label is associated to at most one version label, and therefore specify the set of paragraphs as a function from paragraph labels to version labels:

```
type PARAGRAPHS is FUN(PAR,VERSION)
```

5.1 The Change Types

Now let us turn to the "lower" specification of the history of requirements changes. Each particular history will be represented by an element of type `HISTORY`, with formal type parameters `PAR` and `VERSION`:

```
type HISTORY(PAR,VERSION)
```

The different change types are specified inductively as generators of this type. In the specification, the declarations of these generators follow after the directive `generators`.

The start of a new change history, before any changes have been done, is represented by the generator `root_`, declared

```
    root_: -> HISTORY
```

For convenience, we will use the underscore as a postfix for the generators and functions in the lower specification. The declaration signifies that the generator `root_` is a *constant* of type `HISTORY`, i.e. it does not assume any arguments. Accordingly, we can use `root_` to represent a new history, where no changes have been made yet.

The addition of a new paragraph to a given history is represented by the generator `add_`, declared

```
    add_: HISTORY x PAIR(PAR,VERSION) -> HISTORY
```

The first argument is the given history, and the second argument the new paragraph. In a similar way, the deletion of an existing paragraph is represented by the generator `delete_`, declared

```
    delete_: HISTORY x PAIR(PAR,VERSION) -> HISTORY
```

By splitting a given paragraph, we can replace an existing paragraph by a set of new paragraphs. Accordingly, three arguments are needed, i.e. the existing history, the existing paragraph, and the set of new paragraphs. This is specified with the generator `split_`, declared

```
    split_: HISTORY x PAIR(PAR,VERSION) x PARAGRAPHS -> HISTORY
```

The three change types for combining existing paragraphs, deriving a new paragraph, and replacing a set of paragraphs all require three arguments, i.e. the existing history, the given set of existing paragraphs, and the new paragraph. These change types can be represented by three generators `combine_`, `derive_`, and `replace_`, declared

```
    combine_, derive_, replace_:
        HISTORY x PARAGRAPHS x PAIR(PAR,VERSION) -> HISTORY
```

Finally, the modification of an existing paragraph involves changing its version. This can be specified with the generator `modify_`, declared

```
    modify_: HISTORY x PAR x VERSION**2 -> HISTORY
```

Since each paragraph will be associated to at most one version, `modify_` could alternatively be specified

```
    modify_: HISTORY x PAR x VERSION -> HISTORY
```

The different generators of the type `HISTORY` are then:

```
generators
```

```
root_: -> HISTORY
add_, delete_: HISTORY x PAIR(PAR,VERSION) -> HISTORY
split_: HISTORY x PAIR(PAR,VERSION) x PARAGRAPHS -> HISTORY
combine_, derive_, replace_:
    HISTORY x PARAGRAPHS x PAIR(PAR,VERSION) -> HISTORY
modify_: HISTORY x PAR x VERSION**2 -> HISTORY
```

Note that any change history can be specified by a term constructed by composition of these generators. (Not all terms of type `HISTORY` represent a legal history, but that is taken care of in the upper layer specification, see section 5.11). Suppose that the paragraph labels are given by `p1`, `p2`, `p3`, ..., and the version labels by `v0`, `v1`, `v2`, ... The development leading up to, and including, the introduction of paragraph `p4` is then represented by the term

```
combine(
  modify_(
    modify_(
      add_(
        add_(root_,pr(p1,v0)),
        pr(p2,v0)),
      p1,v0,v1),
    p2,v0,v1),
  ins(ins(emp,pr(p1,v1)),pr(p2,v1)),pr(p4,v0)).
```

5.2 Animation

For animation purposes, we will represent the scenario in Figure 2 by a constant `history_case`, declared

```
history_case: -> HISTORY
```

and defined

```
history_case ==
    modify_(
      modify_(
        replace_(
          ...,
            add_(
              add_(root_,pr(p1,v0)),
              pr(p2,v0)),
          ...,
          ins(ins(emp,pr(p7,v2)),pr(p8,v1)),
          pr(p13,v0)),
        p11,v0,v1),
      p12,v0,v1).
```

The term representing the history looks rather awkward, and we will therefore use HALDEN Prover's conversion mechanism to simplify the output. A conversion procedure for type HISTORY can be implemented as a Prolog predicate. If the specification of HISTORY is written on the file ‘history.sp’, and the conversion program on the file ‘history-c.pl’, then the conversion procedure is automatically consulted when loading the specification into the HALDEN Prover. The automatic conversion of terms is turned on and off by means of the option ‘Eval - Conversion’. With conversion on, `history_case` is output by the HALDEN Prover as

```
P: history_case
```

```
P is
root
add     (p1,v0)
add     (p2,v0)
add     (p3,v0)
modify  (p1,v0) -> (p1,v1)
modify  (p2,v0) -> (p2,v1)
modify  (p3,v0) -> (p3,v1)
combine (p1,v1),(p2,v1) -> (p4,v0)
modify  (p3,v1) -> (p3,v2)
add     (p5,v0)
modify  (p4,v0) -> (p4,v1)
derive  (p3,v2),(p5,v0) -> (p6,v0)
split   (p4,v1) -> (p7,v0),(p8,v0),(p9,v0),(p10,v0)
modify  (p6,v0) -> (p6,v1)
modify  (p7,v0) -> (p7,v1)
delete  (p9,v0)
modify  (p10,v0) -> (p10,v1)
modify  (p6,v1) -> (p6,v2)
modify  (p7,v1) -> (p7,v2)
modify  (p8,v0) -> (p8,v1)
add     (p11,v0)
combine (p10,v1),(p6,v2) -> (p12,v0)
replace (p7,v2),(p8,v1) -> (p13,v0)
modify  (p11,v0) -> (p11,v1)
modify  (p12,v0) -> (p12,v1)
```

The output is now easy to read, as it is straightforward to identify the different (chronological) steps in the change history.

5.3 The Initial Paragraphs

The change type *add* is obviously of basic importance to a change history since it introduces new paragraphs without any reference to existing paragraphs. The paragraphs thus introduced therefore play the important role as initiators for the development of paragraphs. It is important to note that the initial paragraphs are typically replaced during the course of the development, and are therefore not necessarily included in the final set of applicable paragraphs.

The set of initial paragraphs can easily be specified by means of a function that takes a change history as argument and returns the set of paragraphs introduced by the change type *add*. From the stated purpose of the function, it is easy to see that the function `initial_` must have the following declaration:

```
initial_: HISTORY -> PARAGRAPHS
```

No paragraphs are introduced in an empty change history, and the first equation is therefore

```
initial_(root_) == emp.
```

A change of type *add* introduces an initial paragraph, and the equation for this change type therefore inserts this paragraph into the set returned from the recursive call:

```
initial_(add_(H,P)) == ins(initial_(H),P).
```

For the other change types, the equation simply constitutes the recursive call. By way of example, the equation of change type *delete* is

```
    initial_(delete_(H,_)) == initial_(H).
```

Based on the complete definition of `initial_`, we can find the initial paragraphs from the given change history by

```
> initial_(history_case).

P: initial_(history_case)

    P is [pr(p11,v0),pr(p5,v0),pr(p3,v0),pr(p2,v0),pr(p1,v0)]
```

5.4 Deleted Paragraphs

Similarly to finding the initial paragraphs, we can easily specify the deleted paragraphs in terms of a function `deleted_`, declared

```
    deleted_: HISTORY -> PARAGRAPHS
```

The function is defined like `initial_`, except that the deleted paragraphs constitute the set of paragraphs to be returned:

```
    deleted_(root_) == emp.
    deleted_(add_(H,_)) == deleted_(H).
    deleted_(delete_(H,P)) == ins(deleted_(H),P).
    deleted_(split_(H,_,_)) == deleted_(H).
    ...
```

We can then find the deleted paragraphs in the given change history by:

```
> deleted_(history_case).

P: deleted_(history_case)

    P is [pr(p9,v0)]
```

Similar functions can of course be introduced for all the different change types.

5.5 The Applicable Paragraphs

An important aspect of a change history is the set of applicable paragraphs. Since a paragraph may be deleted, replaced, etc., not all paragraphs introduced necessarily survive. Modifications also imply that a new version of a paragraph may be introduced to replace the existing version. Finding the set of applicable paragraphs is in general a non-trivial problem, especially when the number of paragraphs is large or many changes are undertaken. In order to solve this problem, we will introduce a function that, given a change history, returns this set. Accordingly, the function `paragraphs_` is declared

```
    paragraphs_: HISTORY -> PARAGRAPHS
```

That is, the function takes a change history as argument and returns a set of applicable paragraphs. This set will be given by the definition, to be developed in the following based on the recursive structure of the generator terms.

Obviously, the set of paragraphs of an empty change history is empty, and so

```
    paragraphs_(root_) == emp.
```

Changes of type *add* introduce a new paragraph, and the equation for this change type therefore simply inserts the new paragraph into the set returned from the recursive call:

```
    paragraphs_(add_(H,P)) == ins(paragraphs_(H),P).
```

Analogously, the equation for changes of type *delete* becomes

```
    paragraphs_(delete_(H,P)) == del(paragraphs_(H),P).
```

Equations for the other change types may be a bit more complicated as they involve more than one paragraph. By way of example, splitting a paragraph involves deleting the old paragraph and introducing the new ones:

```
    paragraphs_(split_(H,P,S)) == union(del(paragraphs_(H),P),S).
```

Similar equations are given for the other change types. Based on the complete definition of `paragraphs_`, we can find the applicable paragraphs from the given change history by

```
> paragraphs_(history_case).

P: paragraphs_(history_case)

    P is [pr(p12,v1),pr(p11,v1),pr(p13,v0),pr(p5,v0),pr(p3,v2)]
```

The result reveals that only five paragraphs are applicable, and shows the paragraph and version labels for each of these. The other paragraphs have been combined, split, deleted, or replaced during the development of the paragraphs.

5.6 The History of a Paragraph

A paragraph has its own history in the sense that it may go through creation, modification, combination, etc., corresponding to the different change types. By way of example, the following fragment from Figure 2 shows the part of the change history that relates to paragraph `p6`:

$$\begin{tikzcd}
 & (p3,v2) \arrow[dr] & & (p5,v0) \arrow[dl, "derive"] \\
 & & (p6,v0) \arrow[d, "modify"] & \\
 & & (p6,v1) \arrow[d, "modify"] & \\
(p10,v1) \arrow[dr] & & (p6,v2) \arrow[dl, "combine"] & \\
 & (p12,v0) & &
\end{tikzcd}$$

Figure 8. The change history of paragraph p6.

Since we represent the different change types in terms of the generators of type `HISTORY`, the history of a given paragraph can easily be specified by recursion on these generators. In this way, the effect of each change type on the history of the paragraph can be specified independently of the other change types.

The fragment of a change history related to a given paragraph is specified with the function `history_`, declared

```
history_: HISTORY x PAR -> HISTORY
```

The definition of history_ consists of eight equations, one for the empty history and one for each change type. Any fragment of an empty history is empty, and we therefore define

```
history_(root_,_) == root_.
```

A change of type *add* relates to a given paragraph if and only if it is this paragraph that is added. This is specified by means of an if-then-else construct:

```
history_(add_(H,P),I) ==
    if fst(P)=I
    then add_(history_(H,I),P)
    else history_(H,I)
    endif.
```

As can be seen from the recursive nature of the equation, the rest of the history related to the given paragraph is defined by recursion on the overall history. The other equations are similar, the main difference being the test that checks whether a change relates to the given paragraph. By way of example, a change of type *combine* relates to the paragraph if either the paragraph is (1) one of the combined paragraphs, or (2) identical to the paragraph that is introduced as the result of the change:

```
history_(combine_(H,S,P),I) ==
    if or(I?dom(S),I=fst(P))
    then combine_(history_(H,I),S,P)
    else history_(H,I)
    endif.
```

Given a complete definition of history_, the fragment of history that relates to the paragraph p6 in the example is found by evaluating the term `history_(history_case,p6)` with the HALDEN Prover:

```
> history_(history_case,p6).

P: history_(history_case,p6)

    P is
    root
    combine (p3,v2),(p5,v0) -> (p6,v0)
    modify  (p6,v0) -> (p6,v1)
    modify  (p6,v1) -> (p6,v2)
    combine (p10,v1),(p6,v2) -> (p12,v0)
```

It should be easy to see the correspondence to the change history fragment shown above. Note that the result returned from applying `history_` maintains the relative (chronological) ordering of the different changes.

5.7 Backwards Traceability

Given a set of paragraphs, we want to find the development of paragraphs that leads to these paragraphs, i.e. the minimum fragment of the change history that has influenced the development of given paragraphs. In the following, we will refer the given set of paragraphs of interest as the set *S*. The result of this backwards traceability search is specified recursively with the function `backward_`, declared

```
    backward_: HISTORY x PARAGRAPHS -> HISTORY
```

For any set *S*, tracing backwards in an empty history leaves an empty history:

```
    backward_(root_,S) == root_.
```

A change of type *add* should be included in the backwards tracing if and only if the paragraph added is member of *S*:

```
    backward_(add_(H,P),S) ==
        if P?S
        then add_(backward_(H,S),P)
        else backward_(H,S)
        endif.
```

For most of the other change types, the equations are a bit more complicated as they involve more than one paragraph. By way of example, changes of type *split* involve a paragraph that is split into a set of new paragraphs. If one or more of these latter paragraphs are members of *S*, the split paragraph must be included in the backwards tracing. This gives the following equation:

```
    backward_(split_(H,P,T),S) ==
        if or(P?S,not(inter(T,S)=emp))
        then split_(backward_(H,ins(S,P)),P,T)
        else backward_(H,S)
        endif.
```

Note that this equation also covers the case where the split paragraph is a member of *S*.

In contrast, the equation for changes of type *combine* needs to distinguish between three cases:

- the new paragraph is member of *S*;
- neither the new paragraph nor any of those combined are members of *S*;
- one or more of the combined paragraphs are members of *S*.

This gives the following structure of the equation:

```
    backward_(combine_(H,T,P),S) ==
        if P?S
        then ...
        else if inter(T,S)=emp
             then ...
             else ...
             endif
        endif.
```

In the first case, the change should be included. Furthermore, the combined paragraphs need, from this point on, to be included in the set of paragraphs of interest since they contribute to the development of a paragraph of interest. We therefore get:

```
        ...
        if P?S
        then combine_(backward_(H,union(S,T)),T,P)
        ...
```

The whole equation becomes

```
    backward_(combine_(H,T,P),S) ==
        if P?S
        then combine_(backward_(H,union(S,T)),T,P)
        else if inter(T,S)=emp
             then backward_(H,S)
             else combine_(backward_(H,S),T,P)
             endif
        endif.
```

Similar equations are given for the other change types. Finding the backwards traceability from the paragraph p12, version v1, can now be found by:

```
> backward_(history_case,ins(emp,pr(p12,v1))).

P: backward_(history_case,ins(emp,pr(p12,v1)))

    P is
    root
    add     (p1,v0)
    add     (p2,v0)
    add     (p3,v0)
    modify  (p1,v0) -> (p1,v1)
    modify  (p2,v0) -> (p2,v1)
    modify  (p3,v0) -> (p3,v1)
    combine (p1,v1),(p2,v1) -> (p4,v0)
    modify  (p3,v1) -> (p3,v2)
    add     (p5,v0)
```

```
modify  (p4,v0) -> (p4,v1)
derive  (p3,v2),(p5,v0) -> (p6,v0)
split   (p4,v1) -> (p7,v0),(p8,v0),(p9,v0),(p10,v0)
modify  (p6,v0) -> (p6,v1)
modify  (p10,v0) -> (p10,v1)
modify  (p6,v1) -> (p6,v2)
combine (p10,v1),(p6,v2) -> (p12,v0)
modify  (p12,v0) -> (p12,v1)
```

This corresponds to the following fragment of the example change history:

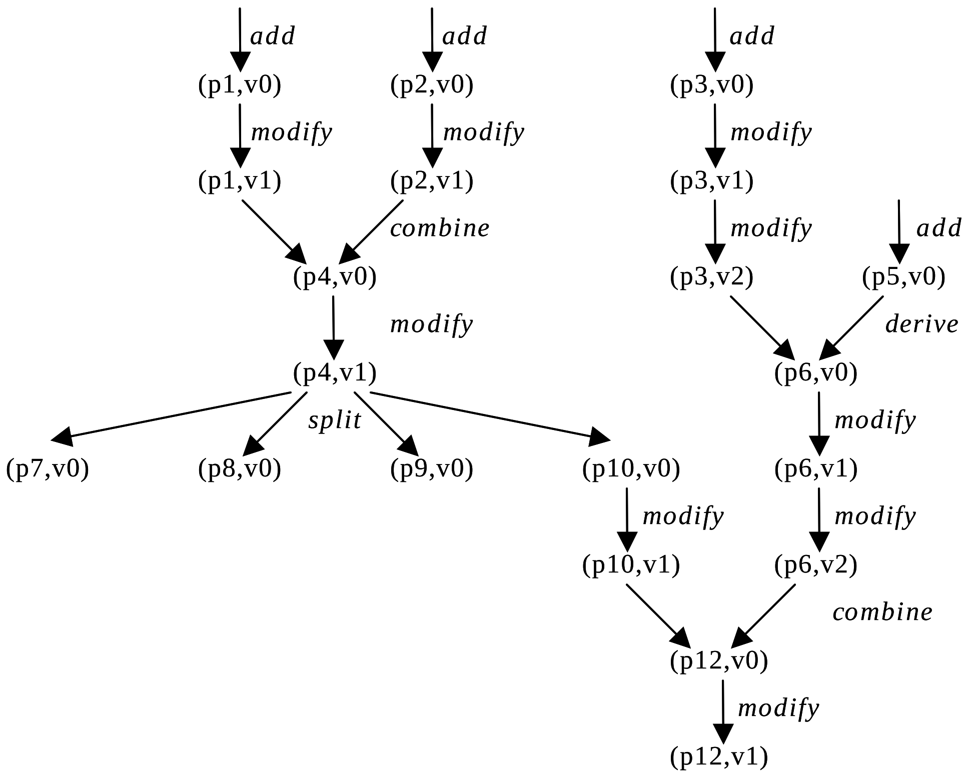

Figure 9. The backwards traceability from paragraph p12, version v1.

5.8 Forwards Traceability

Forwards traceability relates to the development of paragraphs starting with one or more of the paragraphs in a given set of paragraphs of interest. Since recursion on elements of type `HISTORY` involves tracing back through the history, our strategy is to trace back to the very beginning and, on our way forwards again, include the paragraphs that are introduced in the relevant changes. The forwards traceability will be specified with the function `forward_`, declared

```
forward_: HISTORY x PARAGRAPHS -> PAIR(HISTORY,PARAGRAPHS)
```

The pair returned from an application of `forward_` includes the relevant fragment of history and the set of paragraphs consisting of *S* and those that were introduced in the further development starting from *S*.

For any set of paragraphs, tracing forwards in an empty history leaves an empty history:

```
    forward_(root_,S) == pr(root_,S).
```

The other equations include a recursive call that returns the rest of the relevant history, and the possibly extended set of paragraphs of interest. In the case of `add_`, this gives

```
    forward_(add_(H,P),S) ==
        F is forward_(H,S) in
          ...
```

A change of type *add* is included if and only if the paragraph added is included in the (possibly extended) set of paragraphs of interest. The change does not involve other paragraphs, and the resulting equation is

```
    forward_(add_(H,P),S) ==
        F is forward_(H,S) in
          if P?snd(F)
          then pr(add_(fst(F),P),snd(F))
          else F
          endif.
```

Again, the equations for most of the other change types are a bit more complicated as they involve more than one paragraph. By way of example, the equation for changes of type *split* need to distinguish between three cases:

- the old paragraph is member of the given set of paragraphs;
- neither the old paragraph nor any of those introduced are members of the given set;
- one or more of the introduced paragraphs are members of the given set.

This gives the following equation

```
    forward_(split_(H,P,T),S) ==
        F is forward_(H,S) in
          if P?snd(F)
          then pr(split_(fst(F),P,T),union(snd(F),T))
          else if inter(T,snd(F))=emp
               then F
               else pr(split_(fst(F),P,T),snd(F))
               endif
          endif.
```

Similar equations are given for the other change types. Finding the forwards traceability from the paragraph p3, version v0, can now be found by:

```
> fst(forward_(history_case,ins(emp,pr(p3,v0)))).

P: fst(forward_(history_case,ins(emp,pr(p3,v0))))

    P is
    root
    add     (p3,v0)
    modify  (p3,v0) -> (p3,v1)
    modify  (p3,v1) -> (p3,v2)
    derive  (p3,v2),(p5,v0) -> (p6,v0)
    modify  (p6,v0) -> (p6,v1)
    modify  (p6,v1) -> (p6,v2)
    combine (p10,v1),(p6,v2) -> (p12,v0)
    modify  (p12,v0) -> (p12,v1)
```

This corresponds to the following fragment of the example change history:

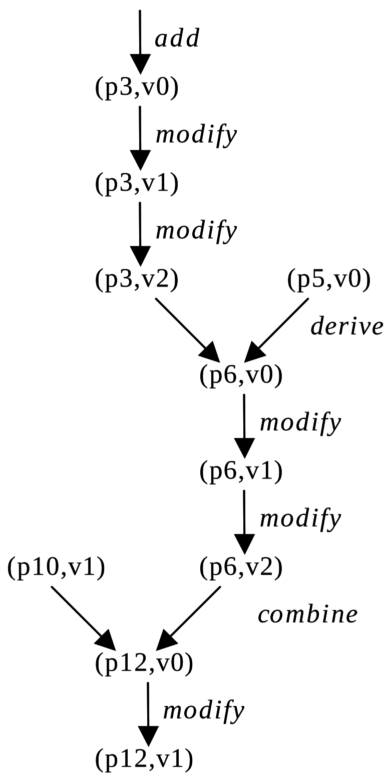

Figure 10. The forward traceability from paragraph p3, version v0.

5.9 Legality of Changes

So far, the specification has not been concerned with the problem of *legality* of a change. Legality should not be confused with the validity of changes, discussed in section 4.1. While validation concerns the semantics of the changes, the legality of a change can be checked mechanically from the structure of the change history tree.

As was briefly mentioned when introducing the generators, not all generator terms necessarily represent a legal history. In order to distinguish the legal changes, we will introduce some predicates that define the conditions that determine whether or not a change is legal. Common to these predicates is that they take as arguments a set of paragraphs and the arguments to the associated change type (except from the change history, since the necessary information is provided by the set of paragraphs). By way of example, the legality of a change of type *add* will be specified with the function `add_ok`, declared

```
    add_ok: PARAGRAPHS x PAIR(PAR,VERSION) -> BOOL
```

The one requirement we give to the legality of a change of this type is that the paragraph is not already present in the set of paragraphs. Since the paragraphs are represented as a function from paragraph labels (`PAR`) to version labels (`VERSION`), this corresponds to checking whether the given paragraph label is a member of the domain of this function. This gives the following definition of `add_ok`:

```
    add_ok(T,P) == not(fst(P)?dom(T)).
```

In a similar way we introduce a predicate `delete_ok` for changes of type *delete*, declared

```
    delete_ok: PARAGRAPHS x PAIR(PAR,VERSION) -> BOOL
```

The definition is indeed very simple. The only requirement is that the given predicate is a member of the set of predicates:

```
    delete_ok(T,P) == P?T.
```

The definitions of the other predicates may be somewhat more complicated as they involve more than one paragraph. By way of example, the predicate for changes of type *split* is declared:

```
    split_ok: PARAGRAPHS x PAIR(PAR,VERSION) x PARAGRAPHS -> BOOL
```

(Note the similarity to the declaration of the corresponding generator). For a change of type *split* to be legal, we require that the given, existing predicate is member of the set of predicates, and that the new predicates are not already represented (in some version or another). Furthermore, we require that the set of predicates introduced is non-empty. This latter requirement can be included by using two equations, one for each case. Thus we get the following definition:

```
    split_ok(_,_,emp) == false.
    split_ok(T,P,ins(R,F)) ==
        and(P?T,
            inter(dom(ins(R,F)),dom(T))=emp).
```

Similar predicates can be given for the other change types. These predicates can now be used to check the legality of any proposed change. By way of example, adding the paragraph (p5,v1) to the example history is illegal since the paragraph (p5,v0) already exists:

```
> add_ok(paragraphs_(history_case),pr(p5,v0)).

P: ...

    P is false
```

In the next subsections, we will see how `add_ok` and the other predicates can be utilized in different ways.

5.10 Legal Change Histories

The predicates determining legality, and the recursive structure of the generator terms (representing the change histories), can be utilized in the specification of what constitutes a legal change history. A change history is legal if all the changes are legal. This can be specified with a function legal_, declared

```
    legal_: HISTORY -> BOOL
```

and defined by recursively calling the appropriate predicate. An empty change history is trivially legal, hence

```
    legal_(root_) == true.
```

The remaining equations all have the same general form, based on the observation that a change history is legal if and only if all previous changes *and* the last change are legal. For changes of type *add* this gives the following equation:

```
    legal_(add_(H,P)) ==
        and(legal_(H),
            add_ok(paragraphs_(H),P)).
```

The other equations are similar. By way of example, the equation of changes of type *split* is

```
    legal_(split_(H,P,S)) ==
        and(legal_(H),
            split_ok(paragraphs_(H),P,S)).
```

It is now very easy to check whether a given change history is legal. Suppose that `illegal_history_case` is identical to `history_case`, except that paragraph (p5,v0) has not been added:

```
> legal_(illegal_history_case).

P: ...

    P is false
```

If we want to find out which of the proposed changes that is illegal, we can facilitate this by specifying a function `legal_find` that returns `root_` if the change history is legal, otherwise the change history up to and including the most recent illegal change. The function is declared

```
    legal_find: HISTORY -> HISTORY
```

and defined by utilizing the predicates defined above. By way of example, the equation for changes of type *add* is:

```
    legal_find(add_(H,P)) ==
        if add_ok(paragraphs_(H),P)
        then legal_find(H)
        else add_(H,P)
        endif.
```

The other equations are similar. By applying this function on `illegal_history_case`, we easily find the most recent illegal change. Failure to introduce paragraph (p5,v0) causes the derivation of paragraph (p6,v0) to be illegal:

```
> legal_find(illegal_history_case).

P: ...

    P is
    root
    add     (p1,v0)
    add     (p2,v0)
    add     (p3,v0)
    modify  (p1,v0) -> (p1,v1)
    modify  (p2,v0) -> (p2,v1)
    modify  (p3,v0) -> (p3,v1)
    combine (p1,v1),(p2,v1) -> (p4,v0)
    modify  (p3,v1) -> (p3,v2)
    modify  (p4,v0) -> (p4,v1)
    derive  (p3,v2),(p5,v0) -> (p6,v0)
```

5.11 The Upper Layer Specification

In this subsection, we will introduce the upper layer specification of the Traceability Model. The specification will utilize the lower layer specification introduced in the foregoing, but will have some advantages:

- The specification is, considered as a program, more efficient than the lower layer specification, since it represents the applicable paragraphs explicitly.
- The specification of the changes in terms of operations makes it easier to check the legality of a change while building up the change history.
- The consequences of illegal changes can more easily be studied because both these changes, and those dependent on these, are filtered out in the construction of the change history.

Instead of building up the change history by explicitly constructing the corresponding generator term, the use of the change operations ensures that only legal change histories are constructed. The specification does however employ the generators, functions, and predicates of the lower layer specification to facilitate this.

Recall that the lower layer specification is given by the specification of the type `HISTORY`. In order to distinguish the upper layer specification from this one, we will introduce a new data type `COLLECTION`, the elements of which are pairs of elements of type `HISTORY` and `PARAGRAPHS`:

```
type COLLECTION(PAR,VERSION) is PAIR(HISTORY(PAR,VERSION),PARAGRAPHS)
```

Intuitively, a collection consists of the change history and the set of applicable paragraphs. Accordingly, the operation for each change type should check that the proposed change is legal and, if so, update the change history and the set of applicable paragraphs accordingly.

For convenience, the operations for the different change types will be specified with functions with identifiers identical to the corresponding generators of type `HISTORY`, except that the postfix underscore is removed. The operation for the initialisation of the system is specified with the function root, declared

```
    root: -> COLLECTION
```

The definition is very simple, as it introduces a collection consisting of an empty change history and an empty set of applicable paragraphs:

```
    root == pr(root_,emp).
```

The general structure of the functions for the other change types reflects that the proposed change is included if and only if it is legal. By way of example, changes of type *add* is specified with the function `add`, declared

```
    add: COLLECTION x PAIR(PAR,VERSION) -> COLLECTION
```

and defined by using the following structure:

```
    add(C,P) ==
        if add_ok(snd(C),P)
        then ...
        else C
        endif.
```

The structure of the definition makes evident that, if the proposed change is not legal, the collection should be left unmodified. Note that `snd(C)` returns the second element of the given pair, here the set of applicable paragraphs. Similarly, `fst(C)` returns the change history. If the change is legal, the change history and the set of applicable paragraphs should be updated:

```
    add(C,P) ==
        if add_ok(snd(C),P)
        then pr(add_(fst(C),P),
                ins(snd(C),P))
        else C
        endif.
```

Note that the modification of the set of applicable paragraphs corresponds to the definition of paragraphs_ in the specification of `HISTORY`. The other change types are specified in a similar way. By way of example, changes of type *split* are specified with the function `split`, declared

```
    split: COLLECTION x PAIR(PAR,VERSION) x PARAGRAPHS -> COLLECTION
```

and defined

```
    split(C,P,S) ==
        if split_ok(snd(C),P,S)
        then pr(split_(fst(C),P,S),
                union(del(snd(C),P),S))
        else C
        endif.
```

The term of type `COLLECTION` corresponding to the constant `history_case`, is declared

```
    collection_case: -> COLLECTION
```

and defined in precisely the same way, except that the underscores have been removed:

```
    collection_case ==
        modify(
          modify(
            replace(
              ...,
                add(
                  add(root,pr(p1,v0)),
                  pr(p2,v0)),
              ...,
              ins(ins(emp,pr(p7,v2)),pr(p8,v1)),
              pr(p13,v0)),
            p11,v0,v1),
          p12,v0,v1).
```

Evaluating the term gives a pair consisting of the change history and the set of applicable paragraphs corresponding to `history_case`:

```
> collection_case.

P: collection_case

    P is
    fst:
    root
    add     (p1,v0)
    add     (p2,v0)
    add     (p3,v0)
    modify  (p1,v0) -> (p1,v1)
    modify  (p2,v0) -> (p2,v1)
    modify  (p3,v0) -> (p3,v1)
    combine (p1,v1),(p2,v1) -> (p4,v0)
    modify  (p3,v1) -> (p3,v2)
    add     (p5,v0)
    modify  (p4,v0) -> (p4,v1)
    derive  (p3,v2),(p5,v0) -> (p6,v0)
    split   (p4,v1) -> (p7,v0),(p8,v0),(p9,v0),(p10,v0)
    modify  (p6,v0) -> (p6,v1)
    modify  (p7,v0) -> (p7,v1)
    delete  (p9,v0)
    modify  (p10,v0) -> (p10,v1)
    modify  (p6,v1) -> (p6,v2)
    modify  (p7,v1) -> (p7,v2)
    modify  (p8,v0) -> (p8,v1)
    add     (p11,v0)
    combine (p10,v1),(p6,v2) -> (p12,v0)
    replace (p7,v2),(p8,v1) -> (p13,v0)
    modify  (p11,v0) -> (p11,v1)
    modify  (p12,v0) -> (p12,v1)
    snd: [pr(p12,v1),pr(p11,v1),pr(p13,v0),pr(p5,v0),pr(p3,v2)]
```

The equivalence between the change history produced by `collection_case` and the generator term represented by `history_case` can be demonstrated directly:

```
> fst(collection_case)=history_case.

P: ...

    P is true
```

In a similar way, we can demonstrate that the set of paragraphs produced by `collection_case` is identical to the set of paragraphs found by searching through `history_case`:

```
> snd(collection_case)=paragraphs_(history_case).
```

```
P: ...

    P is true
```

What about illegal change histories? If we let `illegal_collection_case` correspond to `illegal_history_case` (introduced above), we find that evaluation gives different results:

```
> fst(illegal_collection_case)=illegal_history_case.

P: ...

    P is unknown
    Failed in proving:
      modify_(...,p11,v0,v1)=modify_(...,p12,v0,v1)
```

An important difference between `illegal_collection_case` and `illegal_history_case` is that, while the latter is a generator term, the first is a composition of functions that filters out illegal changes. Since one change filtered out may influence the legality of subsequent changes, the evaluation of `illegal_collection_case` returns the largest possible legal sub-history:

```
> illegal_collection_case.

P: illegal_collection_case

    P is
    fst:
    root
    add     (p1,v0)
    add     (p2,v0)
    add     (p3,v0)
    modify  (p1,v0) -> (p1,v1)
    modify  (p2,v0) -> (p2,v1)
    modify  (p3,v0) -> (p3,v1)
    combine (p1,v1),(p2,v1) -> (p4,v0)
    modify  (p3,v1) -> (p3,v2)
    modify  (p4,v0) -> (p4,v1)
    split   (p4,v1) -> (p7,v0),(p8,v0),(p9,v0),(p10,v0)
    modify  (p7,v0) -> (p7,v1)
    delete  (p9,v0)
    modify  (p10,v0) -> (p10,v1)
    modify  (p7,v1) -> (p7,v2)
    modify  (p8,v0) -> (p8,v1)
    add     (p11,v0)
    replace (p7,v2),(p8,v1) -> (p13,v0)
    modify  (p11,v0) -> (p11,v1)
    snd: [pr(p11,v1),pr(p13,v0),pr(p10,v1),pr(p3,v2)]
```

This corresponds to the following figure:

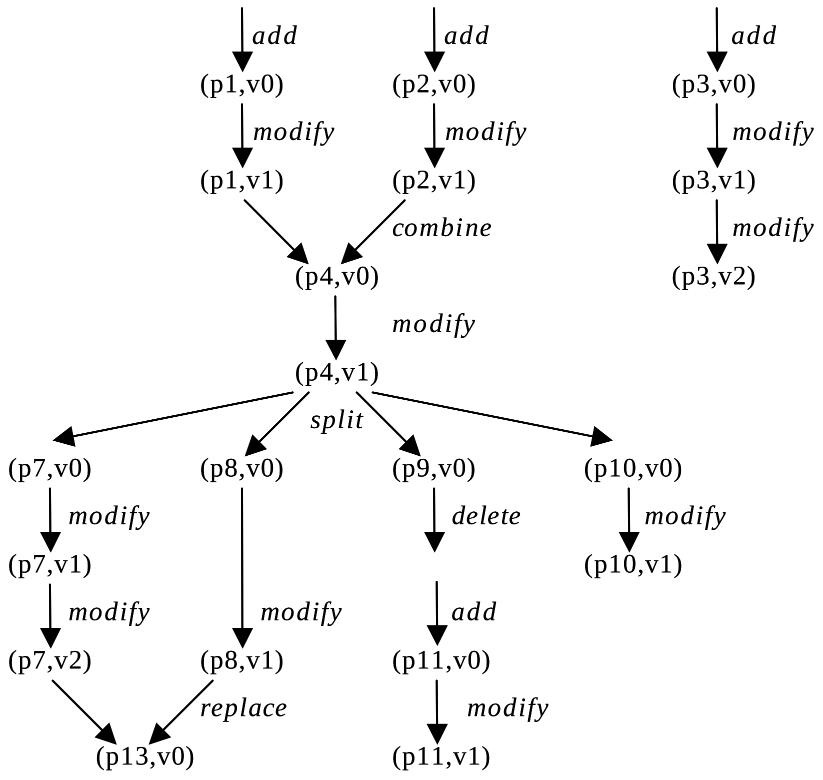

Figure 11. The largest possible legal sub-history when paragraph p5 is not introduced.

The different kinds of analysis specified for `HISTORY` can easily be included in the specification of `COLLECTION`. By way of example, backward and forward traceability can be specified with the functions backward and forward, respectively, declared

```
backward, forward: COLLECTION x PARAGRAPHS -> HISTORY
```

and defined:

```
backward(C,P) == backward_(fst(C),P).

forward(C,P) == fst(forward_(fst(C),P)).
```

Before we conclude, let us see how we can formally capture the intended relationship between elements of type `HISTORY` and `COLLECTION` by means of a so-called *abstraction function*. The purpose of this function is to map elements of type `HISTORY` to the corresponding elements of type `COLLECTION`. The function `absHISTORY` is declared

```
absHISTORY: HISTORY -> COLLECTION
```

and defined

```
absHISTORY(root_) == root.
absHISTORY(add_(H,P)) == add(absHISTORY(H),P).
absHISTORY(delete_(H,P)) == delete(absHISTORY(H),P).
...
absHISTORY(modify_(H,J,V,W)) == modify(absHISTORY(H),J,V,W).
```

Applying `absHISTORY` on the term `illegal_history_case` effective produces the same change history as `illegal_collection_case`:

```
> absHISTORY(illegal_history_case)=illegal_collection_case.

P: ...

    P is true
```

6. References

[1] C. Ghezzi, M. Jazayeri, D. Mandrioli, *Fundamentals of Software Engineering*, 2nd edition (Prentice-Hall, 2003).

[2] E. Hull, K. Jackson, and J. Dick, *Requirements Engineering* (Springer-Verlag, 2002)

[3] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques* (Wiley, 1998).

[4] P. Lindsay and O. Traynor. Supporting fine-grained traceability in software development environments. Technical Report No. 98-10, Software Verification Research Centre, School of Information Technology, The University of Queensland (July 1998).

[5] A. Saeed, R. de Lemos, and T. Anderson, On the safety analysis of requirements specifications for safety-critical software, *ISA Transactions*, **34** (1995), pp. 283-295.

[6] T. Sivertsen, A case study on the formal development of a reactor safety system, in: *Proceedings of FME'96: Industrial Benefit and Advances in Formal Methods*, Volume **1051** of *Lecture Notes in Computer Science* (Springer-Verlag, 1996) pp. 18-38.

[7] T. Sivertsen, Putting principles into practice: the formal development of a theorem prover, in: *Proceedings of 26th Water Reactor Safety Information Meeting*, United States Nuclear Regulatory Commission, Washington DC (1998).

[8] T. Sivertsen, Algebraic specification of an inductive theorem prover, in: *Proceedings of IEEE TC-ECBS and IFIP WG10.1 3rd Joint Workshop on Formal Specifications of Computer-based Systems* (2002).

[9] T. Sivertsen, Undefinedness vs. underspecification in HALDEN ASL, *Nordic Journal of Computing*, **11** (2004), pp. 12-34.

7. Appendix A: Project Organisation and Activities

7.1 Project Organisation

The project is coordinated by Terje Sivertsen (IFE), and comprises the following organisations and persons:

Organisation	Address	Project participants
IFE	Institute for energy technology P.O. Box 173 NO-1751 Halden Norway	Terje Sivertsen +47 69 212403 (terje.sivertsen@hrp.no)

		Rune Fredriksen +47 69 212430 (rune.fredriksen@hrp.no) Atoosa P-J Thunem +47 69 212322 (atoosa.p-j.thunem@hrp.no)
VTT	VTT Industrial Systems P.O. Box 1301 FIN-02044 VTT Finland	Olli Ventä +358 20 722 6556 (olli.venta@vtt.fi) Janne Valkonen +358 20 722 6469 (janne.valkonen@vtt.fi) Jan-Erik Holmberg +358 20 722 6450 (jan-erik.holmberg@vtt.fi)
Ringhals AB (Barsebäck Kraft)	Barsebäck Kraft P.O. Box 524 SE-246 25 Löddeköpinge Sweden	Jan-Ove Andersson +46 46 724148 (jan-ove.andersson@ringhals.se)

The project coordinator is responsible for organising the work within the project and for directing it towards its objectives. This includes

- project planning and tracking;
- establishment and maintenance of the project archive;
- establishment of good communication and cooperation within the project;
- reporting to NKS;
- coordination of activities, in particular the production of the project deliverables;
- follow up of meetings and decisions;
- securing of proper quality control, including review and approval of documents included in the project archive;
- reporting of deviations and implementation of agreed corrections.

All the individual participants represent important parts of the technical competence within the project, and are responsible for contributing to the activities in such a way that the project reaches its objectives.

The project organisation is intended to constitute a Nordic expert network on requirements elicitation, specification, and assessment for digital I&C. The network will provide a forum for exchanging experiences and research results on the questions to be addressed by the project, and will provide a basis for evaluating the relative merits of the different practices, the relative importance of identified criteria, etc. A related concern is to facilitate knowledge transfer from other areas applying equipment that are used in NPPs.

The emphasis on best practices and identified success criteria means that the project needs to deal with real cases involving the development of a digital I&C system. By organising the project on basis of a Nordic expert network, the project contributes to the synthesis of knowledge and experiences, enhancement of competence on requirements elicitation, specification, and assessment, improved awareness of alternative practices, a basis for assessing current practices, and an incentive to search for best practice.

7.2 Project Activities and Further Plans

The project is carried out through a combination of project meetings, industrial seminars, and coordinated report writing. Each meeting focuses on a limited set of issues, where the participating organisations are asked to prepare a presentation on their experiences and viewpoints. Particular emphasis is given on concrete experiences from safety-critical applications. On this basis, the meetings attempt to provide a synthesis, evaluate the merits of the different practices, etc. The project activities in 2004 have included two regular project meetings and one industrial seminar, participation at the Nordic Seminar on Automation, and participation at the EHPG meeting of the OECD Halden Reactor Project. At both of the latter two meetings, a summary of the TACO project was presented, thereby reaching a wide international audience. The minutes from the industrial seminar are given in appendix B.

The TACO project, which was initiated in 2002, delivered a preproject report in January 2003. The purpose of the preproject report was first of all to provide a technical basis and plan for further work. Particular emphasis was put on relating knowledge on relevant software engineering issues to NPP needs and practice. In this sense the report, with its identification of challenges and issues, provides an adequate basis and reference point for more detailed discussions and evaluations.

The activities in 2003 constituted a natural continuation of the preproject, and focused on the technical issues concretised in the preproject report. The work concentrated on four central and related issues, viz.

- Representation of requirements origins
- Traceability techniques
- Configuration management and the traceability of requirements
- Identification and categorisation of system aspects and their models

The work was presented at the first TACO Industrial Seminar, which took place in Stockholm on the 12th of December 2003. The seminar was hosted by SKI.

On basis of this work and responses received at the Industrial Seminar, the TACO activity in 2004 started by preparing a contribution to the Nordic Seminar on Automation, Oskarshamn, 5-7 April 2004. The TACO activity was also presented at the EHPG meeting of the OECD Halden Reactor Project, in Sandefjord, Norway, on the 10th of May 2004. The further activity focused on providing a unified exposition on the issues studied and thereby facilitate a common approach to requirements handling, from their origins and through the different development phases. The approach was presented at the second TACO Industrial Seminar, which took place in Helsinki on the 8th of December 2004. The seminar was hosted by STUK.

The responses received at the second Industrial Seminar will be utilized in the writing of the final report of the TACO project, to take place in the first half of 2005. The project is scheduled for completion on the 30th of June 2005.

The discussions from the meetings and the progress of the project are carefully reported by means of detailed minutes.

The overall documentation schedule is as follows:

- January 2003: Preproject report (*completed*)
- December 12th, 2003: Presentations and materials to the first TACO Industrial Seminar (*completed*).
- January 5th, 2004: Documentation of the work for 2003 collected and sent in a suitable form to NKS (*completed*).
- April 5th, 2004: Presentations and materials to the Nordic Seminar on Automation (*completed*).
- December 2004: Presentations and materials to the second TACO Industrial Seminar (*completed*).
- January 5th, 2005: Documentation of the work for 2004 collected and sent in a suitable form to NKS (*the present report - completed*).
- June 30th, 2005: Final TACO project report.

8. Appendix B: Minutes from the Second TACO Industrial Seminar

Where: *STUK, Laippatie 4, Helsinki*
When: *Wednesday 8th December 2004, 9:00 - 16:00*
Chairman: *Terje Sivertsen*
Secretary: *Janne Valkonen*

Participants

Øivind Berg	Institute for energy technology	Norway
Rune Fredriksen	Institute for energy technology	Norway
Terje Sivertsen	Institute for energy technology	Norway
Atoosa P-J Thunem	Institute for energy technology	Norway
Bo Liwång	SKI	Sweden
Jan-Ove Andersson	Ringhals AB	Sweden
Erik Wallgren	Ringhals AB	Sweden
Christer Fransson	Oscarshamn Kraftgrupp AB	Sweden
Peter Bertilsson	Oscarshamn Kraftgrupp AB	Sweden
Karl-Erik Eriksson	Oscarshamn Kraftgrupp AB	Sweden
Olli Ventä	VTT Industrial Systems	Finland
Björn Wahlström	VTT Industrial Systems	Finland
Jan-Erik Holmberg	VTT Industrial Systems	Finland
Janne Valkonen	VTT Industrial Systems	Finland
Sixten Norrman	VTT Processes	Finland
Harri Heimburger	STUK	Finland
Petteri Tiippana	STUK	Finland
Pauli Suvanto	STUK	Finland
Juha Halminen	TVO	Finland
Petteri Lehtonen	Fortum	Finland

The intention with the Second TACO Industrial Seminar was to present and discuss the work within the TACO project with a wider audience representing actors in the nuclear sector in the Nordic countries. In this way, the seminar would contribute to the dissemination of the research results to the intended end-users, and also to providing input to further work within the follow-up project MORE - Management of Requirements in NPP Modernization Projects.

8.1 Agenda

09:00	Welcome
09:15	Finnish Radiation and Nuclear Safety Authority perspective, Petteri Tiippana, STUK
09:45	Introduction to the TACO project, Terje Sivertsen, IFE
10:15	Relationship to other activities, Björn Wahlström, VTT
10:45	The TACO Common Approach to Requirements Management, Terje Sivertsen, IFE
11:30	Safety I&C system reliability; requirements, verification and life cycle, Petteri Lehtonen, Fortum
12:00	Lunch
13:00	Active workshop on requirements management in Nordic modernization projects and possibilities for industrial utilization of the TACO deliverables.
16:00	Adjourn

8.2 Welcome, *Harri Heimburger (STUK) & Terje Sivertsen (IFE)*

Harri Heimburger welcomed everybody and explained some practical issues. After that, TACO project manager Terje Sivertsen expressed his satisfaction about the large number of participants from the different Nordic countries, and gave special thanks to Harri Heimburger for his positive and helpful response to the idea of arranging the second industrial seminar in the premises of STUK. Sivertsen emphasised the importance of the seminar as a source of feedback and direction of the future work, introduced the agenda, and called attention to the interactive workshop which was to be held in the afternoon.

8.3 Finnish Radiation and Nuclear Safety Authority perspective, *Petteri Tippana (STUK)*

In his presentation, Petteri Tiippana focused on requirements in a new plant project. He explained the procedure of setting requirements for a nuclear power plant and how this procedure was implemented for Olkiluoto 3. Tiippana presented the life-cycle of requirements in a plant project and also the expectations of the project from different viewpoints. He also described the regulatory review process. The vendor verifies to STUK that safety requirements are managed in the project, from PSAR via detailed design to the input given to the manufacturer. This involves that the safety requirements can be demonstrated to be correctly transformed through the chain. This is closely related to requirements traceability and the contract agreement about the vendors’ responsibilities with respect to requirement management. Tiippana concluded by emphasising the importance and difficult nature of requirements management as an activity.

Questions and answers:

On a question on what makes requirements engineering difficult from a tool perspective, Tiippana (and also Pauli Suvanto) answered that a tool makes requirements engineering more structured but the tools do not provide all the features that are needed. Tools make requirements management more formal and explicit, whereas the guides are more abstract. Tools may not be necessary if you have good management systems, but then requirements management is more dependent on human beings.

It was also commented that one can't think of requirements as a glue because requirements from different parties are different and you have to take that into account. The question was raised whether this would not make it difficult to make a common basis. Tiippana answered that there is a point where everybody's (all stakeholders) requirements meet. Especially in the nuclear island it must be seen that all the parties are working for the same goal and all the requirements should originate from the same safety goals. All parties are working on different levels, but the owner has a special responsibility to ensure that the right requirements are given, and to harmonize the different requirements from the different parties. In essence, the different parties must be working with the same requirements.

On a question whether STUK uses the same tools for all purposes, Tiippana answered that no requirements management tools as such are used at the moment. The only tool used is their own Excel-based tool which is for their own purposes only.

8.4 Introduction to the TACO project, *Terje Sivertsen (IFE)*

In his introduction to the TACO project, Terje Sivertsen gave an overview on the background of the project, some of the main issues, and the project activities 2003-2005.

The background of the project comes from the two following facts: Clear, complete and stable requirements from the beginning are very important for a successful project. Critical software defects tend to be introduced in the early phases of the development chain.

Among the main issues, Sivertsen mentioned several aspects of requirements, requirements engineering, and management, with particular emphasis on the aspects of traceability and communication. Sivertsen maintained that these aspects were central to many software engineering activities and life cycle phases.

Traceability is important in particular for demonstrating correctness of the implementation, but also for tracing back to the original requirements and information sources when needed.

Communication plays an important role for example in demonstrating that the requirements correctly reflect the safety analysis of the plant and other relevant information. With regard to TACO activities, Sivertsen emphasized the role of the two industrial seminars as a way to present the TACO work and extend the network to a broader representation of Nordic nuclear power. The TACO final report will be finished by the 30th of June 2005.

Sivertsen also called attention to the follow-up activity MORE (Management of Requirements in NPP Modernization Projects), scheduled for startup on the 1st of July 2005. In that connection, he invited the seminar participants to consider a closer coupling to the project, thereby helping facilitate the industrial utilization of the TACO deliverables.

8.5 Relationship to other activities, *Björn Wahlström (VTT)*

Wahlström started his presentation by describing what is the problem with digital I&C. He explained what we know about it and what kind of difficulties there are. He also described the structure of digital I&C and the development process of digital I&C. There are several technical and practical problems in the area and new methods and tools are needed. That is just what TACO is doing.

Wahlström described the role and purpose of the IAEA Technical Working Group NPPCI (Nuclear Power Plant Control and Instrumentation) which operates within the framework of the International Atomic Energy Agency. The role of the TWG-NPPCI is to assist the IAEA in identifying and initiating activities enabling the organizations and personnel working with instrumentation and control to make the best use of the available and emerging technologies to meet the plant operational and safety needs in an economic manner. He also introduced some of the recent IAEA reports and activities. The COMPSIS project, some EU activities and American utility requirements were also discussed in the presentation.

One of the points of most relevance to the TACO project related to systems of requirements, and what Wahlström saw as a need for hierarchical structure and more levels within the system of requirements. Another challenge for the future was computerized tools for all phases of design and construction.

Questions and answers:

There was a comment, explained with a whiteboard drawing, which emphasized the question how the platform actually fulfils the requirements (if we have the waterfall model). Wahlström answered that usually some existing platform is taken as a basis. The challenge is to show that the requirements are covered by the platform, and that other functions provided have no negative impact on the intended functionality. There was a comment that one should utilize backwards engineering in order to analyze software. A further comment was that this is a tough challenge. There is one platform and it is difficult to make platforms to fit.

8.6 The TACO Common Approach to Requirements Management, *Terje Sivertsen (IFE)*

Sivertsen introduced the TACO Shell, which is a framework for traceability and communication of requirements. He also introduced the TACO Traceability Model, which facilitates traceability by representing requirements changes in terms of a change history tree. It facilitates also introduction, changes and relationships between different requirements, design steps, implementations, documentation, etc. The change types are creating, deleting, splitting, combining, replacing, deriving, and modifying (without changing the meaning).

He showed a complete example of forwards and backwards traceability in a traceability tree. The change history tree can also be extended to show statements of requirements, show requirements origins, facilitate configuration management, etc.

Sivertsen also introduced plans for the follow-up project MORE (Management of Requirements in NPP Modernization Projects), which aims at improving the means for managing large amounts of evolving requirements in Nordic NPP modernization projects. It will facilitate industrial utilization of the TACO deliverables, including the TACO Traceability Model. It will study how requirements can be properly structured by using the concepts of design patterns and requirements templates, generated by utilizing the change history trees of the TACO TM.

Questions and answers:

It was asked what is the difference between creating and deriving in a traceability tree. Sivertsen answered that you can e.g. derive a requirement from a larger part of other requirements. The derived requirement may not be shown very clearly before that. It is like when decomposing design into smaller modules, thereby making implicit requirements more clear and explicit. Creating is the only change type that makes something from the scratch.

There was also a comment concerning the V-model and traceability. First you have the plant level requirements and then you have the I&C requirements and testing, validation, verification etc. on the right side. Traceability between the different sides of the V-model is important. You should have some kind of a baseline after each step in the left side. Sivertsen explained that this can be facilitated by adopting adequate extensions to the change history tree, similarly to what was exemplified in the presentation.

Another question was whether the model has been concretized as a tool or guidelines. Sivertsen answered that TACO TM has not yet been implemented, but that it is indeed feasible, and that parts of the functionality of such a tool already have been specified within the project.

Another comment was that the TACO TM seems to be better than the usual traceability matrix.

8.7 Safety I&C System Reliability Requirements - Verification - Life cycle, *Petteri Lehtonen (Fortum)*

Lehtonen introduced the modernisation schedule of the Loviisa NPP I&C. Things that will be renewed include: control rooms, operational I&C systems, safety I&C systems, process computer and training simulator. Field instrumentation, cabling, switchgear and severe accident management systems will mainly not be renewed. The next thing he introduced was his Master's Thesis: Verification of Reactor Protection System reliability where he identifies the process of licensing safety I&C systems, establishes reliability requirements for safety I&C and finds the verification means. He also showed a few examples and concluded with the following observations: Reliability requirements can be traced back to life cycle, it enables us to understand better and manage the licensing process and gives us confidence of succeeding in licensing in a very strict time perspective without production losses.

8.8 Active workshop on requirements management in Nordic modernization projects and possibilities for industrial utilization of the TACO deliverables.

8.8.1 Table discussions on the MORE project, *Moderator Atoosa P-J. Thunem*

Input from other projects:
Several stressed the importance of studying and evaluating past and existing projects, especially those with focus on modernization. In that respect, gathering information on successful and not-so-successful efforts was highlighted. Updated information on the CEMSIS project will be sent to the project participants.

Applicability of the TACO results:
There was a discussion on how the TACO Traceability Model (TM) helps to categorise the requirements. The response was that although the TM itself does not categorise the requirements, the inclusion of all requirements involved, the formal definition of the change types and the mapping facilities associated are believed to contribute to easier and more adequate categorisation in general.

The discussion was also around criteria for categorisation, where especially the verifiability of the requirements, the stage in focus during the life cycle, the dependability factors in focus and the application system itself were addressed as important factors. It was explained that the TM will help to develop configurable life cycle models, where specialised portions and versions of the common TM can be used at different life cycle stages, while the common TM itself can provide useful information on which stages are most relevant with regard to the project/task in focus. It was added that the TACO report for 2004 will touch this issue.

A question was raised concerning whether the analysis methods are covered in the TACO project and how the results can contribute to, e.g., reliability analysis. The response was that the TACO results do not focus on a specific dependability factor, but focus on providing means to deal with different types of dependability factors, such as reliability, safety, security, availability and maintainability. This, due to the observation that these factors should not be treated separately but be incorporated into the very nature of the requirements.

In order to evaluate the applicability of the results in a trustworthy manner, it was strongly suggested to gather information from simple but real examples, and proceed from there to more complex case studies. The EUR (European Utility Requirements) was proposed as a good source to begin with.

It was not recommended to use examples from projects that already are terminated, as getting in touch with the suppliers and other parties involved will then become very difficult, hence reducing the amount of input and assistance needed.

Handling complexity/large amounts of requirements:
The choice of language/notation/method was mentioned as one deciding factor. The discussion participants brought up own and others experiences, especially when it comes to the application of formalism and formal methods. The response was that the focus of the project is not to advocate a specific language or method, but to better manage the requirements in mod-

ernisation projects. Therefore, all available equipment relevant to the problem will be considered, which also includes combining several approaches.

From a specific experience (Oskarshamn 1), it was mentioned that different departments by the plant owner, in co-operation, developed the requirements to a certain level, and then further developed to a safety concept by one of the suppliers. These requirements were then further developed and adopted to the technology and the specific platform, by the other supplier. This means that groups with participants of different backgrounds were working during different stages of the project's development process (life cycle). In that regard, one should keep in mind that the parties involved in such projects often could have mutually different ideas about the requirements. This practically means that an apparently common set of functional requirements could result in different sets of software requirements. Therefore the review of the requirements and the specific application, performed by the plant owner, is a very important and critical activity. The role of the platform chosen and knowledge about its capabilities (or the lack of such) are areas that once again were pointed out as examples of requirements that could be interpreted in different ways.

One important factor with regard to handling large amounts of requirements was whether the TM can help identifying "reusable elements" in project development processes. This, due to the observation that many tasks in different projects are in fact the same, and thus there ought to be ways to identify similarities, so that it won't be needed to go through every single task each time a project is established.

There was a discussion about the reasoning mechanisms behind the TM, and how TM can help reasoning about the requirements themselves and their validity. It was explained that although the TM itself does not provide reasoning about the nature of the requirements and their validation, it indeed contributes to doing so, by means of formalism behind the definition of the change types. This, together with the mapping facilities associated with the TM will force the requirement engineer to be more alert on validating the requirements, with regard to their original sources and their consequences.

It was in that regard also suggested taking fundamental issues on the definition of requirement types into consideration. It was pointed out that it is possible to define these types in a mathematical manner ("X should be in place", or "X should be better than Y").

The importance of communication between different parties and groups of people involved as well as the awareness around possible misunderstandings and misinterpretations were also stressed.

8.8.2 Table discussions on Industrial Experiences, *Moderator Olli Ventä*

Is it about large amount of requirements?

- Depends on what you regard as a requirement (but in general yes, even huge).
- There are many levels of requirements, or, things to take into account.
- There are many types of requirements: product, process, system, …
- There are many sources of requirements.

Can one freeze requirements?

- Some freezing points exist: preproject, contract between vendor and utility, several stages during the project.
- The work advances by milestones.
- We never have the time and money for the ideal waterfall model.
- Actually a project iterates through project phases, redoing most of the requirements analysis work.
- Safety-related requirements should be treated differently from other requirements.
- It is hard to separate between pure requirements and implementation.
- Requirements and designs are linked, one works on details on both sides in parallel.
- Choice of platforms, and its limitations, affects the separation between requirements and design.
- Even standards and guides actually mix these things.
- One never starts from an empty table, reuse is heavily utilized.
- Requirements must be allocated to proper/due places in the process, design hierarchies, etc. Establish a framework from the beginning.
- Should formal systems/tools be adopted for requirements management?
- Requirements management is usually well organized, but in general no single formalism is fully adopted.
- Requirements are hard and difficult to maintain with number of items (thousands of identifiers).
- One has to cope with several analyses and verifications with all the material.

New project vs. modernization

- A lot of requirements re-engineering work.
- The more you change (safety concept, I&C functions) the more extra requirements are due.
- Does today's technology help or make it possible formalizing natural language documents?
- It is believed it is easier to start a new plant!

Quality of requirements management results in easier licensing?

- It makes sense, supported by textbooks, and some empirical studies support it.
- You build the self-confidence in the requirements analysis stage, which is then easy to keep.
- Requirements should communicate to all stakeholders, aiming at a common understanding. But such a common understanding is difficult to achieve, and perhaps does not exist at the moment. Different stakeholders may have different targets for requirements analysis. It will not be achieved if not planned to happen.
- The more you have planned in the beginning the easier it is at the end.
- Quality of requirements management also results in easier maintenance in plant operation.

8.8.3 Table discussions on Requirements Change Management, Traceability, Version Control, etc., *Moderator Rune Fredriksen*

There is really no reason for requirements to change if they are defined properly the first time. In fact - requirements should not change at all. This rather philosophical approach invited a discussion on whether we allow - or plan to allow - requirements to change too often. Due to time or budget constraints a “preliminary” set of requirements is often included in the development process - resulting in the need for extensive requirements change.

An experience related to requirements change, is that the problems more often occur in small modernization projects, not in the bigger projects. Small changes do not trigger the necessary processes to discover the consequences of requirements change, and will as a consequence be handled rather informally. Backwards traceability is one approach that is seldom applied in smaller projects. These smaller projects however share a property with the bigger projects. Projects have a tendency to influence each other. One small, apparently insignificant change in a project might have severe influence on another project.

Backwards traceability is not always possible due to the extensive amount of tacit knowledge in an organisation. Requirements that are implicit due to environmental issues, informal work procedures or personal knowledge have not been recorded, and limit the possibilities for tracing back to the original assessment of a requirement.

Another problem arises when small changes that do not seem important at the time, are emerging as important later in the process. Due to the difference in time there are often other people involved at a later stage in the process.

It is easier than ever to change software. This has both positive and negative implications. However - it should perhaps be easier to change a requirement. Changeability of requirements is a property that would make requirements management easier - if there exists a proper way to do this.

It is easier to degenerate the total structure of a system today. Systems used to be large constructs of hardware that in itself was a barrier to change. Today software has substituted much of the design, and since software is easy to change - so is the structure (or design) of the system.

The solution “design for change” looks promising, but might also contribute to complexity. Increased complexity is not something we want in a system. It is important to remember that all software changes are design changes. Software changes are not maintenance.

The basis for good requirements management is a good document system. If you know when, why and how changes are made, the possibility of establishing a correct picture of the system is increased. Without this information you will have to make empirical studies of the system.

Tools provide structure. There exists a need for a tool providing traceability. Commercial tools do provide to a certain degree the possibility of tracing requirements, but are not used extensively.

The following general problems with requirements were stated:

- How do we relate requirements to each other? Can this process be automated?
- How do we minimize requirements change? How do we design for change?
- How do we establish completeness of requirements? Incompleteness is more likely to occur than completeness.

Bibliographic Data Sheet **NKS-103**

Title	Traceability and Communication of Requirements in Digital I&C Systems Development. - Project Report 2004
Author(s)	Terje Sivertsen*, Rune Fredriksen*, Atoosa P-J Thunem*, Jan-Erik Holmberg**, Janne Valkonen**, Olli Ventä** & Jan-Ove Andersson***
Affiliation(s)	* Institute for Energy Technology, Halden, Norway ** VTT, Finland *** Ringhals AB, Sweden
ISBN	87-7893-162-2 *Electronic report*
Date	April 2005
Project	NKS_R_2002_16
No. of pages	54
No. of tables	0
No. of illustrations	11
No. of references	9
Abstract	In 2004, the work has focused on providing a unified exposition on the issues studied and thereby facilitating a common approach to requirements handling, from their origins and through the different development phases. Emphasis has been put on the development of the TACO Traceability Model. The model supports understandability, communication and traceability by providing a common basis, in the form of a requirements change history, for different kinds of analysis and presentation of different requirements perspectives. Traceability is facilitated through the representation of requirements changes in terms of a change history tree built up by composition of instances of a number of change types, and by providing analysis on the basis of this representation. Much of the strength of the TACO Traceability Model is that it aims at forming the logic needed for formalising the activities related to change management and hence their further automation. The work was presented at the second TACO Industrial Seminar, which took place in Helsinki on the 8th of December 2004.
Key words	Traceability, requirements, TACO, change management, digital I&C, systems development