

TRACE: Traceability of Requirements for Analysable Computerised Environments

Atoosa P-J Thunem, Harald P-J Thunem
Institute for Energy Technology, PB 173, NO-1751 Halden, Norway

Abstract

This paper describes a practical approach to dependable requirements engineering of computerised systems, based on the perception that requirements should be identified, specified and implemented for all stages of the system development process (or system lifecycle), and not only for the highest stage. Furthermore, the approach acknowledges the importance of well-defined traceability mechanisms to provide links between the requirements belonging to a particular stage or different stages of the lifecycle. Finally, the approach recognises the relationship between how a requirement can be met and how it can be opposed to, due to unexpected or unwanted events. Thus, the approach aims at making a computerised system and its lifecycle analysable with regard to several dependability factors such as safety, security, reliability, flexibility and maintainability. The paper also provides the main elements of a prototype tool supporting the fundamentals of the approach, and hence promising for expansions as well as tailor-made applications.

1. Introduction

Especially within information and communication technologies (ICT) and their applications in different branches, several approaches have been proposed towards a better software development process. Among the most applied is the Rational Unified Process (RUP) that provides a matrix-oriented lifecycle model highly supporting the time aspect of the lifecycle. Here, the road map is formed by two main activity categories: disciplines followed to develop the system and phases related to its life-path. The workload in each phase is decided by the actual discipline in focus: More elaboration phase is required during the design discipline, whereas more construction is needed during the implementation.

Nevertheless, despite the availability of detailed guidelines for sub-activities in each discipline and for the number of iterations in each phase, neither RUP nor any other lifecycle models provide guidelines on how to achieve traceability among phases and disciplines. Also, if system properties are addressed at all, the implied concern is almost

entirely on functional and operational factors, and not other dependability factors such as safety, security, reliability, flexibility and maintainability. To exemplify, there exist no instructions on how the security issues associated with the specific system architecture or application domain can influence the length of a certain phase, or the amount of certain sub-activities during the iterations.

Management of changes is closely related to the maintainability of the development process of a computerised environment and the result (product) of the process, the operational and applied environment itself. Typically, the requirements at each stage of the development process of a 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 or a desire to incorporate technological advances into the development stages (use of new methods, procedures, tools, etc.). The task of managing alteration of the requirements is highly 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.

In order to validate and verify the requirements and their changes in a dependable manner, different analyses are needed as an integrated part of carrying out each stage of the development process. The most important analysis is that of thorough risk analysis with focus on one or several dependability factors that need to be analysed and assessed, before introducing any progress or any change. There is a need for traceability of the requirements related to a specific risk analysis method or process, in accordance with the requirements of system development process and its product a risk analyst is supposed to analyse.

The remainder of the paper explains the main elements of a tool that aims to support traceability of requirements for each stage of computerised systems' development processes, and through the boundaries between stages. Additionally, the tool also provides means for modelling requirements on different dependability factors related to the development process or its product (the system), and thus for better dependability and risk analysis necessary to better management of changes introduced.

2. Basic elements of the TRACE tool

This chapter describes the basic elements of a tool called TRACE that in combination can be used to achieve the objectives described in the Introduction in an efficient and practical manner. The following summarizes the objectives:

- Traceability between the requirements at a particular stage of the system lifecycle
- Traceability between the requirements defined for different stages of the system lifecycle
- Traceability of changing or changed requirements throughout the system lifecycle for better change management
- Traceability of dependability-related requirements throughout the system lifecycle for better dependability analysis and the associated risk analysis

The basic elements of TRACE are *Paragraphs*, *Changes*, *Change Types*, *Links*, *History Trees*, and *Sets*. The remainder of the paper focuses on their description and their applications to achieve the objectives listed above.

2.1 Paragraphs

The traceability approach and associated tool focuses on the concept of *Paragraphs*, which are objects containing the text describing a specific requirement. Paragraphs are associated with the following list of attributes:

<i>id</i>	Automatically generated unique identifier.
<i>label</i>	Textual short label.
<i>version</i>	Version number. A Paragraph can be subject to a number of different Changes, where some will cause the creation of Paragraphs with a new label, and other the creation of Paragraphs with the same label but incremented version number (see description of Change class below).
<i>time</i>	Time of creation.
<i>status</i>	Status attribute (see table below for possible values).
<i>description</i>	Paragraph content, which for e.g. software development will be the textual description of a requirement. The purpose of the traceability approach is to keep a track of all changes to this attribute across different Paragraph versions and across all development phases.
<i>change_in</i>	The change that caused the creation of the Paragraph.
<i>changes_out</i>	List of changes performed on the Paragraph causing the creation of other Paragraphs.
<i>origins</i>	List of paragraph origins. See description of Link (which is the class implementing the concept of origin) below.

The *status* attribute of a Paragraph or a Change can take the following values:

<i>None</i>	Default Paragraph/Change status.
<i>Created</i>	Indicates that the Paragraph is the first in a list of Paragraphs with the same label, but different version numbers. The Paragraph is the result of either a <i>create</i> Change or a Change performed on another Paragraph which creates one or more new Paragraph(s) (<i>derive, split, combine...</i>).
<i>Trace</i>	The Paragraph/Change is part of a trace result, e.g. a backward trace. The Paragraph/Change will be highlighted in the history tree display.
<i>Highlight</i>	The Paragraph/Change is highlighted in the history tree display.
<i>Deleted</i>	The Paragraph has been explicitly deleted (having been subject to the <i>delete</i> Change).

2.2 Changes

The *Change* class contains the properties of a single Change from one or more Paragraphs into one or more Paragraphs. Changes are associated with the following list of attributes:

<i>id</i>	Automatically generated unique identifier.
<i>type</i>	Type of Change (see description of <i>ChangeType</i> class below).
<i>sources</i>	List of input Paragraphs to this Change.
<i>targets</i>	List of output Paragraphs from this Change.
<i>status</i>	Status attribute (see table above).
<i>user_id</i>	The identifier of the user responsible for introducing the Change.
<i>time</i>	Time of Change introduction.
<i>reason</i>	Textual description of the reason for introducing the Change.
<i>basis</i>	The basis for introducing the Change (see table below).

The *basis* parameter is used to provide some description of the basis for applying the Change to one or more Paragraphs:

<i>Method</i>	The Change has been introduced due to the outcome of some analysis method, e.g. a HazOp analysis, which has suggested that the Paragraph(s) must be updated due to some shortcoming.
<i>Expert</i>	The Change has been introduced due to input from some expert (expert judgement).
<i>None</i>	No special basis is given for the Change.

2.3 Change Types

The *ChangeType* class is used to define different types of Changes. The *ChangeType* class is associated with the following list of attributes:

<i>label</i>	Unique label.
<i>para_in</i>	The number of input Paragraphs (possible values are “0”, “1”, “1 or more” and “2 or more”).
<i>para_out</i>	The number of output Paragraphs (same as above).
<i>description</i>	Textual description of change type.
<i>result_status</i>	Status of output Paragraph(s) (see table above).
<i>update</i>	How to update the output Paragraphs label and version (see below).

The *update* value defines how the Paragraph label and version number are determined for a Paragraph resulting from a Change:

<i>No update</i>	The output Paragraph has the same label and version number as the input Paragraph.
<i>New label</i>	The output Paragraph is given a new label.
<i>Increment version number</i>	The version number of the output Paragraph is incremented relative to the input Paragraph.

For use in software development, the default Change types include:

- create
- modify
- combine
- replace
- split
- derive
- delete
- un-delete

An example of a change type is “modify”, where the attribute values are given in the following table:

<i>label</i>	“modify”
<i>para_in</i>	1
<i>para_out</i>	1
<i>description</i>	“This change denotes a modification of the paragraph”
<i>result_status</i>	None
<i>update</i>	Increment version number

Only one Paragraph at a time can be subject to a *modify* Change, and the result is a single Paragraph where the label remains the same, while the version number is incremented.

2.4 Links

In many cases it can be useful to include information regarding the reason for introducing a Paragraph. Examples of this information can be:

- a textual reference from a brainstorming meeting
- an IAEA safety standard, suggesting the introduction of a specific safety function
- a web-page with statistical data showing the potential improvements in system reliability by developing in accordance with certain object-oriented metrics
- a link between a Paragraph in the *implementation* phase and a Paragraph in the *design* phase, indicating that the former fulfils the requirements of the latter

The *origin* attribute of a Paragraph is used to provide information regarding where the *idea* of the Paragraph originated, and it can be a combination of textual descriptions, files, hypertext links, and other Paragraphs. The Link type implements the concept of the origin attribute, and the attributes associated with the Link type are:

<i>link_type</i>	Type of link
<i>string</i>	Textual information

Examples of Links are given in the following table:

A textual link

```
object Link
  link_type: TEXT
  string: "This Paragraph was included due to a discussion at project meeting
in Halden on 2005-04-08"
end
```

A file link

```
object Link
  link_type: FILE
  string: "c:\projects\more\p08-basis.doc"
end
```

A hypertext link

```
object Link
  link_type: HYPERTEXT
  string: "http://standards.ieee.org/catalog/olis/index.html"
end
```

A Paragraph link

```
object Link
  link_type: PARAGRAPH
  string: "PA_002389" (the ID of a particular Paragraph)
end
```

2.5 History Trees

The *HistoryTree* class is used to hold all required information about one history tree, including all Paragraphs and Changes. An example of a history tree is shown in Figure 1. History trees will show the development of a number of Paragraphs as they are subject to Changes, and for software development projects a typical use is to create one history tree for each development phase.

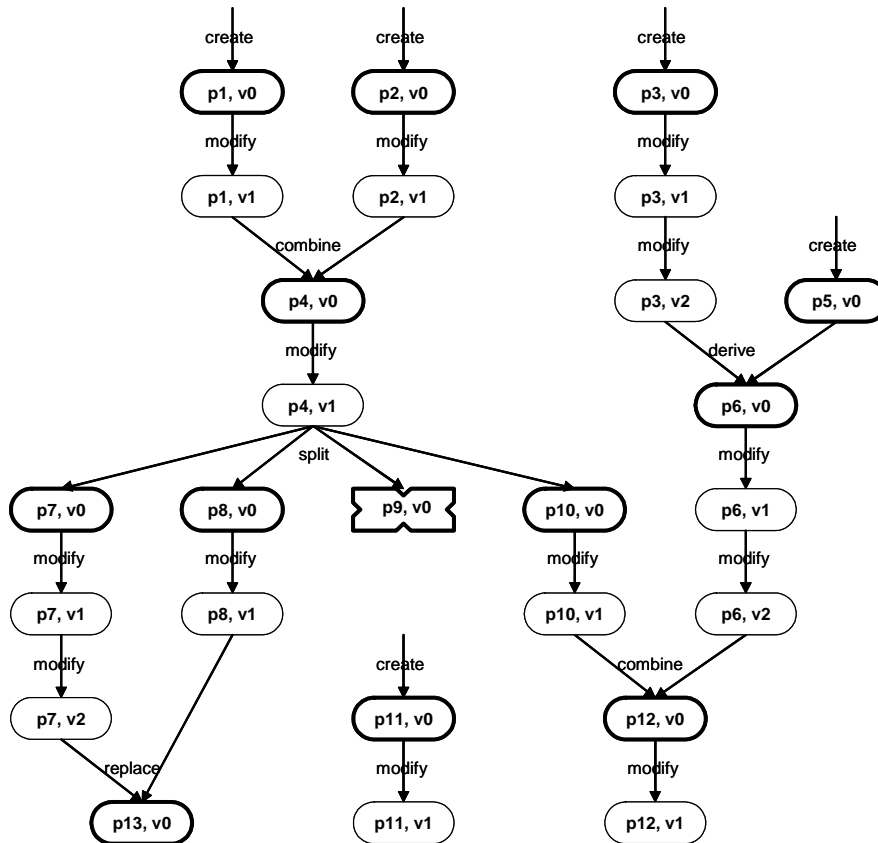


Figure 1. Example history tree.

The list of attributes associated with a HistoryTree is:

<i>id</i>	Automatically generated unique identifier.
<i>label</i>	Textual label provided by user.
<i>paragraphs</i>	List of Paragraphs.
<i>changes</i>	List of Changes.
<i>create_time</i>	Creation time.
<i>last_change_time</i>	Last time history tree was changed.

2.6 Sets

The *Set* class extends the *HistoryTree* class to include a list of subsets, links to parent and child sets, and information about opening and closing times and status. This allows a *Set* to contain any number of *Paragraph* objects, as well as any number of *Set* objects, and to maintain a derivative relationship between *Sets*.

The list of attributes associated with a *Set* (in addition to those inherited from the *HistoryTree* class) is:

<i>sets</i>	List of subsets.
<i>parent</i>	Parent set.
<i>child</i>	Child set.
<i>open</i>	Indicates whether <i>Set</i> is open or closed.
<i>close_time</i>	Time the <i>Set</i> was closed.

One typical use of the *Set* could e.g. be to group all *security-related* requirements into a separate *Set*, facilitating a subsequent *security analysis* and its associated *risk analysis*.

A *Set* will be able to compare its content (specifically its list of *Paragraphs*) to the content of another *Set*, i.e. which *Paragraphs* are common to both *Sets*, and which *Paragraphs* are unique. This ability is particularly relevant in *change management*, where the difference between two versions of the same software with regard to which *Paragraph* versions they implement is readily apparent.

An open *Set* can have its content (i.e. list of paragraphs, history trees and subsets) changed, while a closed set is not editable. In software development this will typically correspond to a version of the software where the feature set has been frozen.

3. Basic analyses

Using the features of the classes described in Chapter 2, the tool can perform a number of analyses relevant to software development and change management:

<i>Created Paragraphs</i>	Whenever a new Paragraph is created, either “from scratch” or by certain Changes to other Paragraphs (e.g. derive, split, combine...), the Paragraph is marked as “Created”.
<i>Current Paragraphs</i>	The current or most recently updated version of a Paragraph is found by iterating through the list of Paragraphs and for each Paragraph label find the Paragraph with the highest version number. (Paragraphs that have been explicitly deleted are not included in this search)
<i>Deleted Paragraphs</i>	Whenever a Paragraph is deleted, it is marked as “Deleted”.
<i>Paragraph History (forward/backward)</i>	The Paragraph history for any Paragraph can be determined by finding all versions of the selected Paragraph, all Changes affecting these versions, as well as the relevant version of all Paragraphs included in these Changes. This is straightforward, as all <i>Paragraph objects</i> contain lists of “incoming” and “outgoing” Changes, and all <i>Change objects</i> contain lists of “input” and “output” Paragraphs.
<i>Paragraph Trace (forward/backward)</i>	<p>Forward: Forward traceability relates to the development of Paragraphs starting with a selected Paragraph. The result will include all Paragraphs affected by the selected Paragraph (see Figure 2).</p> <p>The trace is performed by a recursive search through all output Changes starting with the selected Paragraph. The search through a sub-tree is halted once a Paragraph without any output Changes is reached.</p> <p>Backward: Given a Paragraph, we want to find the development of Paragraphs that leads to this Paragraph, i.e. the minimum fragment of the Change history that has influenced the development of the given Paragraph (see Figure 3).</p> <p>The trace is performed by a recursive search through all input Changes starting with the selected Paragraph. The search through a sub-tree is halted once a Paragraph whose input is a “create” Change is reached.</p>
<i>Origin Trace</i>	The <i>origin</i> parameter in the Paragraph class provides links to information used when creating a Paragraph. This information could e.g. be a textual description of why the Paragraph should be included, a shortcut to a file, a hypertext link to an IEEE standard used as basis for the Paragraph, or a link to another Paragraph in a different history tree. A typical use of the origin parameter could be during a software development project, where a separate history tree is created for each development phase (requirement, design, implementation, test...). Here, each Paragraph would represent a specific version of a specification, and often a specification in the design phase would be based on a specification in the requirement phase (see Figure 4).

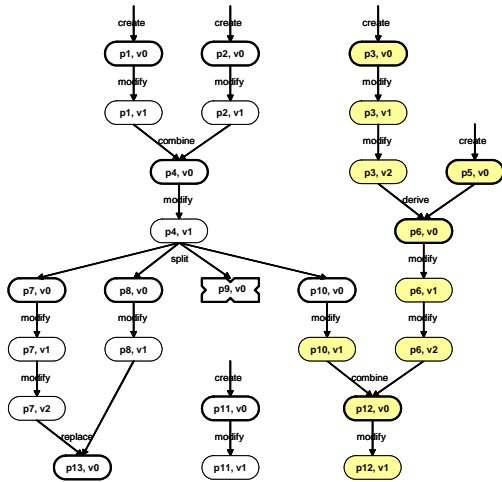


Figure 2. Forward trace from (p3, v0).

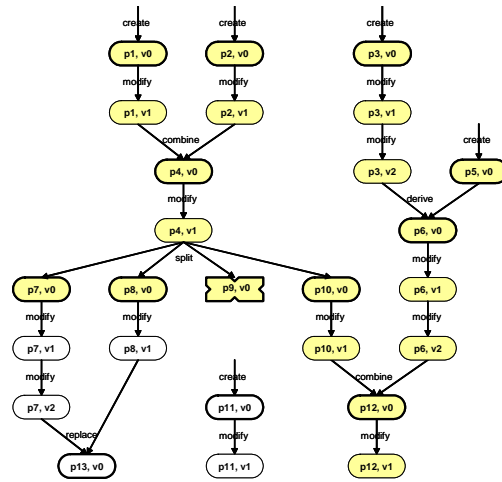


Figure 3. Backward trace from (p12, v1).

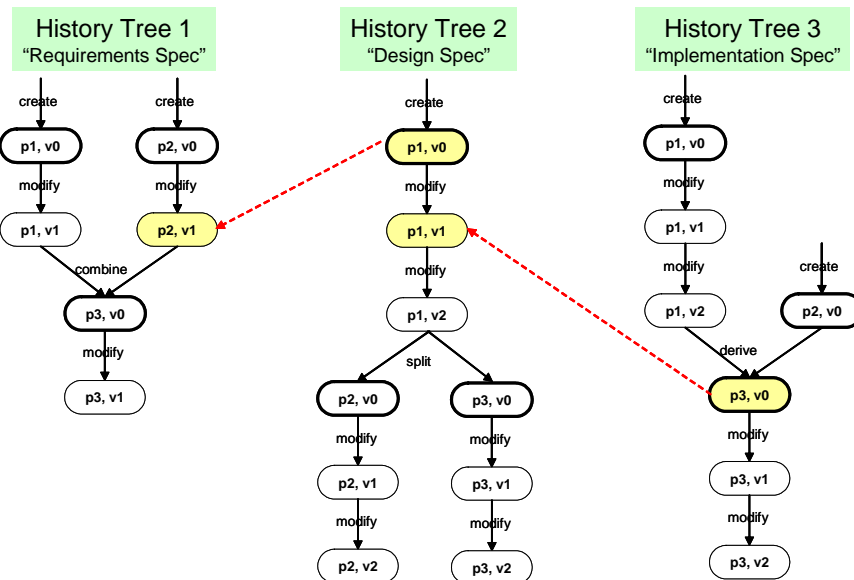


Figure 4. Origin trace. The dotted lines are links from Paragraphs in one development phase to a previous phase.

4. References

1. T. Sivertsen, R. Fredriksen, A.P-J Thunem, J-E. Holmberg, J. Valkonen, O. Ventä and J-O. Andersson: "The TACO Approach for Traceability and Communication of Requirements", SafeComp 2005 (LNCS 3688), pp. 317-329