Systematic Requirements Recycling through Abstraction and Traceability

Antje von Knethen[1], Barbara Paech[1], Friedemann Kiedaisch[2], Frank Houdek[3]

[1]Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6, D-67661 Kaiserslautern
{vknethen, paech}@iese.fhg.de

[2]University of Ulm Faculty of Computer Science
D-89069 Ulm
friedemann.kiedaisch @informatik.uni-ulm.de

[3]DaimlerChrysler AG Research and Technology
P.O. Box 23 60
D-89013 Ulm
frank.houdek @daimlerchrysler.com

Abstract

Ad-hoc recycling between requirements documents of product variants is a major source of requirements defects. In this paper, we present an approach for systematic requirements recycling based on a combination of abstraction (in terms of a template) and traceability (between requirements). The main features of our approach are the use of conceptual models to determine relationships necessary for correct recycling and the focus on minimizing link setting. This approach can also be used to develop abstraction and traceability guidelines tailored to other application domains and traceability goals, such as change or project management.

1. Introduction

In production companies, such as the car industry, many product variants are typically developed in parallel or subsequently, and each product variant is specified in a separate requirements document. Of course, the documents have many similarities. To save effort, developers try to reuse as much requirements as possible between the different variants. Requirements recycling occurs frequently and is often carried out ad-hoc. Such non-systematic approaches are time consuming and error prone. One reason for these problems is that it is difficult to identify requirements that can potentially be reused because of the varying structure of requirements documents. In addition, often not all requirements related to a copied requirement are copied, because these dependencies are not explicit. For the same reason, too many requirements might be copied. Thus, omissions, inconsistencies, and superfluous features are introduced into the new document. At DaimlerChrysler (DC), an effort is being undertaken together with Fh IESE to improve current practice through *systematic requirements recycling*, where recycling means the activity of taking requirements from existing requirements documents and inserting them into a new requirements document that describes a similar product (e.g., because of a new car release of a given type or because of reuse of certain car features between different car types). The goal of the cooperation is to find a simple, yet effective process that would require only minimal additional effort and training of the developers.

The literature distinguishes between two general types of reuse approaches [1]: composition- and generation-based approaches. Generation-based approaches focus on instantiating reusable abstractions. Popular examples are product-line approaches [2] or patterns [3]. Composition-based approaches are based on composing reusable components. This type of approach is typically applied for design or code reuse. For requirements reuse, generation-based approaches are more popular [4]. One exception is [5], which describes an approach for composition-based requirements reuse that classifies requirements to support identification of reusable requirements.

For our purposes, pure product line techniques, such as domain modeling [2][6][7], which create a specification of the commonalities and variabilities of the product line, were not viable. The use of domain models induces a considerable change of the process for the developers, because they will mainly focus on the instantiation of the decision model, not on the creation of a new requirements document. At DC - as is typical for many production companies - it was required that

- the resulting process should only marginally deviate from the existing process,
- the size of the documents should not increase significantly and
- the effort of the developers should not increase significantly.

In addition, there was no time to develop detailed abstractions necessary to document variabilities.

Thus, we looked for an approach that keeps some of the benefits of a product line approach, but adheres to the process and document restrictions. We combined concepts of generation- and composition-based reuse: A template, which includes the entities of the existing documents to be recycled in all future documents, gives the generation-based facet of the approach. This template defines a document structure, captures the commonalities of all requirements documents, and restructures the original document to reduce the number of relationships. The composition-based facet of the approach are the guidelines on how to document relationships of recycling candidates explicitly and on how to use relationships to copy recycling candidates correctly. The traceability guidelines solve the problem of omission and inconsistencies through selective recycling (i.e., copying an entity with small changes).

We developed both, the template and the guidelines, based on a set of typical change scenarios. This helped us identify the documentation entities most likely to be copied. In addition, we developed a traceability model to make the relationships between the requirements explicit. This helped us to restructure the document template and to derive the guidelines. The template and the guidelines were validated against the list of change scenarios and accepted by DC.

The novelty of our approach lies in the combination of abstraction and traceability, in particular in the usage of conceptual models to document commonalities and reduce relationships. Although the requirements documents investigated were on the system level, our approach is equally applicable to pure software documents.

This paper is structured as follows. In Section 2, we describe a typical recycling situation derived from DC passenger car development and we discuss related work in Section 3. Then, we present the foundations of our approach, in particular the traceability model that includes a conceptual system model and documentation model. In the major part of our paper (Section 5 and 6), we present more detailed change scenarios and our solution with its benefits and risks. In addition, we compare our solution with other possibilities. Finally, we summarize our main contribution and future work in section 7.

2. Typical requirements recycling situation

This section illustrates a typical requirements recycling situation. This is influenced by DC passenger car development. Consequently, we use examples that are typical for cars. However, similar situations may be found in various contexts and industries.

Almost always, development activities for a new (sub-) system do not start on the green field but on existing solutions. Typically, functionality is enhanced incrementally. Accordingly, the set of requirements increases, too. The system-level requirements for electronic control units (ECU) for vehicles may comprise documents of several hundred pages. A significant portion of requirements from one increment to another (almost) does not change at all. There are well-engineered functions, like the speedometer, that differ at most in (optical) design details. This naturally leads to reusing old documents when specifying the same ECU for a new car. However, reuse is carried out ad-hoc: the person in charge copies an old document and edits or enhances all parts s/he considers relevant. S/he integrates parts from other documents that deal with functionalities s/he has to add. Obviously, this approach is error prone.

In addition, different persons create the source requirements documents (RDs) over time. Thus, deviations in the RD's structure make it even harder for the engineer to identify related requirements.

In previous projects [8][9], DC tried to restructure existing RD to get rid of unnecessary information and to introduce a common basis for further improvements. But this structure was not very well accepted by the engineers, because the new RD became too large, even if there was no change in content. As in all industries, the workload of the engineers is pretty high. Often, they are not computer scientists, but electrical technicians or other engineers. Their main focus is the car development, not writing advanced RDs. All these reasons limit the overhead accepted with new processes. As stated in the introduction, a solution to the recycling problem must maintain the size of the RD and the workload. Labor-prone rework of RD (e.g., for domain engineering) is not an option.

3. Related work

Supporting reuse through a template is also advocated in [10]. This template supports fine-grained reuse on the level of words in sentences. In contrast, we support coarse-grained reuse of documentation entities. [11] also uses a conceptual model to capture similarities, however, this work focuses on similarities between different domains, while we focus on similarities between documents in the same domain.

From the traceability literature, three general types of relationships can be identified: The first type relates entities in the same software artifact (e.g., a requirement depends on another requirement). These relationships are called *horizontal relationships* [12]. The second type relates entities of different artifacts (e.g., a set of design classes realizes a requirement). These relationships are called *vertical relationships* [12]. The third type relates entities of different versions of an artifact. These relationships are called *evolutionary relationships* [13]. Direct support for requirements recycling requires investigating horizontal relationships (i.e., between entities in one version of a requirements document).

There are many approaches to traceability, but only few to tracing horizontal relationships. To confirm this observation, we conducted a literature survey [14].

Previous studies on and approaches to traceability aimed at understanding and characterizing traceability and at describing general principles on how to implement traceability (e.g., [15][16][17][18][19]). The approaches focus on vertical traceability to support change. Ramesh and Jarke [19] found that most approaches simply specify that there are horizontal relationships without the ability to specify their nature. They pointed out that it is very valuable to distinguish among different types of dependencies and classified dependencies into four categories: goal, task, resource, and temporal dependencies. They did not state which types of dependencies should be traced to support a certain goal, such as recycling.

To get more insight into practical experience with traceability, we screened reports of tool user conferences like InDOORS [20]. These reports confirmed the findings of the study [19] that traceability is mainly used for post-traceability in terms of requirements refinement, requirements allocation and compliance verification, and – for high end users – also, pre-traceability or rationale capture. Again, only few reports on the details of supporting horizontal relationships could be found. One exception is [21], which reports on experience in using DOORS to ensure completeness of the products. In this case, the following relationships are used:

- Cascade links (i.e., evolution or part-of links) between artifacts of the same type (e.g., from requirements for the interior to requirements for the vehicle).
- Interdependency links representing dependencies within one artifact (e.g., requirements for fuel economy and vehicle weight).
- Interface links representing dependencies according to the flow of data, material or energy between elements of different requirements artifacts (e.g., engine and transmission requirements).

In the following section, we describe the relationships that we used to support the recycling goal.

4. Identification of relationships

Software artifacts consist of several parts (e.g., functional requirements, design classes). In the following, we call these parts *documentation entities*.

4.1. Relationships

Documentation entities are related through different types of relationships. Each documentation entity represents a certain *logical entity* (e.g., a paragraph of an "Overview Description" (documentation entity) represents a function "seat control" (logical entity)). We suggest identifying types of horizontal relationships by investigating logical entities described in the RD and their relationships. Of interest are relationships between logical entities but also relationships between logical and documentation entities.

[22][23] distinguishes two general types of relationships between logical entities: (1) refinement and (2) dependency relationships. *Refinement relationships* are relationships between logical entities on different levels of abstraction. A complex system function "seat control", for instance, has a refinement relationship to a set of system functions, such as "seat back angle control". *Dependency relationships* are relationships between logical entities on the same level of abstraction. A system function "seat control", for example, has an influence relationship to an environmental item "seat position".

Relationships between documentation entities can be derived from relationships between logical entities. A section "Seat Control" of an RD, for example, represents the logical system function "Seat Control" (see Figure 1, lower half). One subsection of the same section includes an "Input list". One paragraph describes a "Push button" that represents a logical item "Push button". There is a dependency relationship "monitors" between the logical entities "Push button" and "Seat Control" because pushing the button results in moving the seat. From this logical relationship, a dependency relationship "monitors" between the paragraph that describes the "Push button" and the section "Seat Control" can be derived.

In order to identify all types of horizontal relationships, it is not enough to investigate logical entities and their relationships. Typically, a logical entity is described more than once in a software document, for example, in different views. *Representation relationships* are relationships between two documentation entities that represent the same logical entity [22][23]. If a system function, for instance, is represented in paragraph "Seat Control" within chapter "Product overview" and section "Seat Control" within chapter "Function Description" (see Figure 1, lower half), there is a representation relationship between the section and the paragraph.

To deal with several RDs (as in the case of requirements recycling), one has to abstract from concrete entities and their relationships. The contents of an RD are captured on the level of types. The arrows between the lower and the upper half of Figure 1 illustrate the relationships between concrete and abstract logical and documentation entities.

4.2. Conceptual system model

A *conceptual system model (CSM)* [22][23] describes *types* of logical entities and their relationships that appear in several RDs of a certain domain. Each relationship described for a certain logical entity type (e.g., "function" has a "dependency relationship" to "monitored environmental item") must be instantiated for a logical entity of this type described in a concrete RD (e.g., function “seat control” has a dependency relationship to a monitored environmental item “push button”).

We developed a CSM for electronic control units. The model is based on the Four Variable Model for documenting embedded systems developed by Parnas et al. [24][25]. The model describes the contents of different software artifacts and their relationships without specifying their representation.

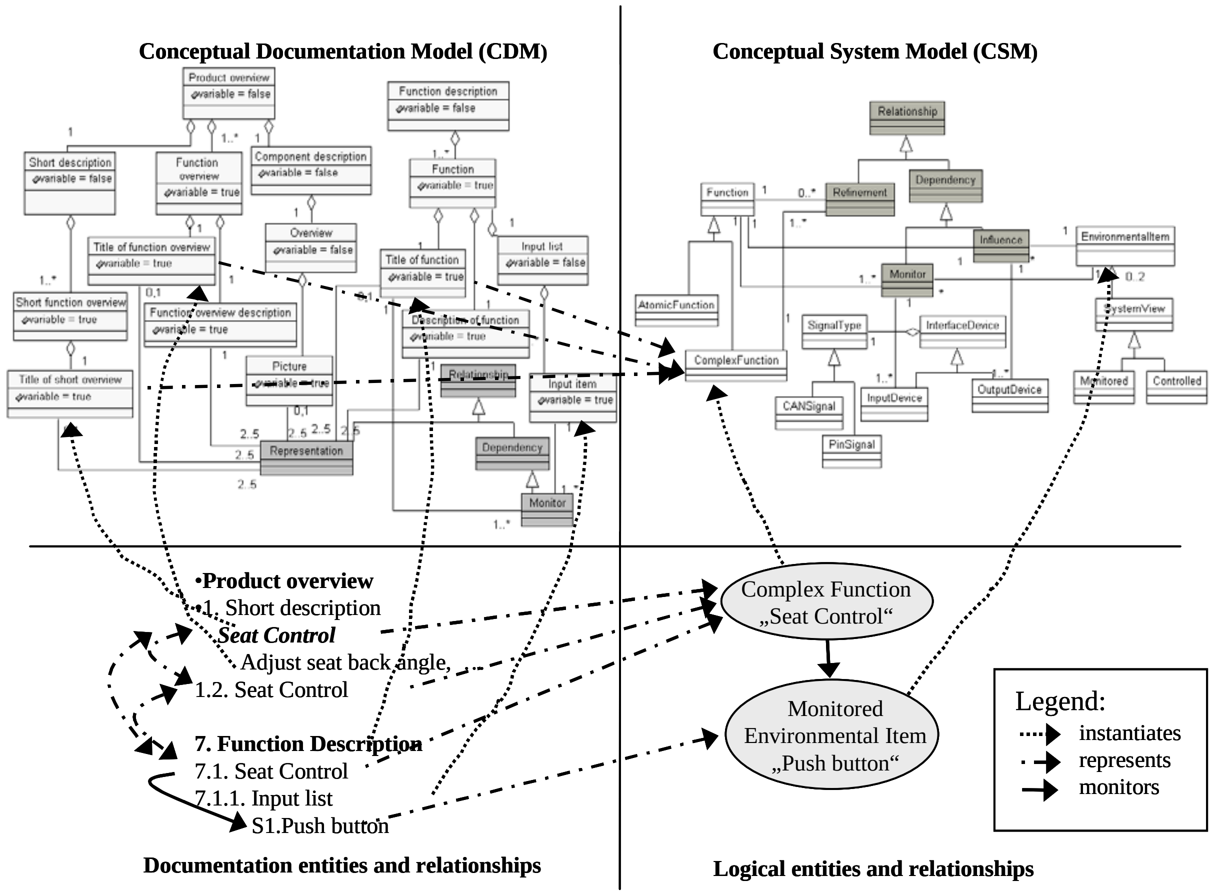

Figure 1: Documentation and logical entities and their types

Figure 1 (upper half) shows a subset of our CSM. The UML class diagram describes logical entity types and relationships of the logical entity type “atomic function”. The different kinds of relationship types are marked in gray. The model describes, for example, a “refinement relationship” between “complex functions” and “functions”, and different dependency relationships. Each “monitor relationship” is, for instance, associated to one or more “input devices” that enable the monitoring of an “atomic environmental item”.

4.3. Conceptual documentation model

A *conceptual documentation model (CSM)* [22][23]) describes *types* of documentation entities and their relationships that appear in several RDs in a certain domain. Each relationship described for a documentation entity type (e.g., a paragraph included in section “Product Overview” has a “representation relationship” to a section of the chapter “Function Description”) must be instantiated for each documentation entity of this type (e.g., the paragraph that describes the function “Seat Control” has a “representation relationship” to a section “Seat Control” of the chapter “Function Description”).

Each documentation entity type has an attribute “variable” that defines whether the documentation entity type must be instantiated with a type “name” in the RD (value = false) or whether it can be instantiated differently (value = true). The type “Function description”, for example, must be instantiated to a heading “Function Description”. In contrast, the type “Input item” can be instantiated to types “Push button”, “Transmitter”, etc.

Figure 1 (upper half) shows a subset of our CDM. The UML class diagram describes documentation entity types (colored light gray) and their representation relationships (colored dark gray). In addition, the figure shows the relationships between the CDM and the CSM (dashed lines) used to identify the representation relationships. As can be seen from the CDM, six documentation entities have a representation relationship because they all represent an "complex function". The figure only visualizes the three dashed "represents" arrows corresponding to the example in the lower half.

Dependency and refinement relationships are transferred from the CSM to the CDM. The figure shows a dependency relationship between a paragraph "Title of Function" of a section "Function" and a paragraph "Input item" of a subsection "Input list".

Our traceability model consists of a CSM and a CDM. Both models are the foundation of our recycling approach described in the next chapter. From this model, we derived guidelines on how to prepare the documents for recycling and how to recycle requirements.

5. Systematic Requirements Recycling

In this section, we describe the systematic process Fh IESE recommended to DC as well as the activities that are necessary to provide the details of the process.

5.1. Change Scenarios

To guide the search for an efficient recycling process, DC identified the most common variation scenarios between two generations of ECUs influencing the recycling:

1.) Changes in the system environment
- New sensors or actuators
- Sensors or actuators, formerly wired directly, now communicate via the CAN-bus.
- Additional messages for communication with other ECUs via the CAN-bus.

2.) Architectural changes
- Functionalities are relocated between ECUs.

3.) Changes in functionality
- New, innovative functionality is integrated into an existing ECU
- Existing functionality is enhanced or redesigned.

4.) Changes of parameters
- Limits for values like voltage, speed, etc. change due to new technology or legal restrictions.

5.) Changes in conditions
- New laws, standards, business rules must be adhered to.
- Variants for specific countries must be handled.

5.2. Systematic Recycling Process

Systematic requirements recycling is based on the notion of *logical recycling candidates.* A logical recycling candidate is a logical entity described in an existing RD that can be used (with minor modifications) in a new RD. A documentation entity representing a logical recycling candidate is called *documentation recycling candidate.*

Our recycling approach combines composition-based and generation-based reuse concepts. It provides a template (generation-based) that

- defines a document structure,
- includes stable entities of an RD, and
- restructures the original document to reduce the number of relationships.

In addition, it provides traceability guidance (composition-based) on

- how to document relationships of recycling candidates explicitly and on
- how to use relationships to copy recycling candidates correctly.

Both, the template and the guidance are based on the traceability model.

Requirements engineers who want to develop a new RD with the help of recycling candidates execute the following process steps:

- Take the template and use its structure as a checklist to ensure completeness of the new RD.
- Search for documentation recycling candidates in the existing RD. In general, this requires a complex matching process. In this context, however, matching is not the problem, because the developers know very well how to identify the recycling candidate. Thus, keyword search is sufficient.
- If documentation recycling candidates were found: Copy these together with related documentation entities into the new RD and adapt them.
- If no recycling candidates were found: Add new documentation entities to the new document and relate them to the rest of the document. The latter supports recycling of the new RD.

5.3. Development of the Recycling Process

The template and the guidelines for the development of a new RD must be tailored to the domain and traceability goal, for instance, change, recycling or project management. In the following, we describe the activities to develop the recycling process.

1.) Distinguish documentation recycling candidates from documentation entities not relevant for new RD. Important recycling candidates are mentioned by the change scenarios. Since the scenarios cannot cover all reuse situations, domain know-how is necessary in addition to identify reusable documentation entities.

2.) Develop a CSM and a CDM of a typical existing RD. Types for all (documentation or logical) recycling candidates and related (documentation or logical) entities are included in the models. Since the models are on an abstract level of types (e.g., “function” instead of “seat control”), the number of entities and, therefore, the number of relationships is limited.

3.) Restructure the CDM to minimize representation relationships and to position documentation entities related by dependency relationships close to each other. Minimization of representation relationships reduces redundant information on a specific logical entity type in the document as much as possible. This supports the identification of documentation recycling candidates as well as the effort during copying. The latter is also supported through the positioning.

4.) Develop a template for the new RD corresponding to the restructured CDM. Each documentation entity type with variable value “false” is instantiated by a (sub-) heading or a fixed text in the template.

5.) Provide guidelines for documenting relationships between documentation entities that reduce the necessity for link setting. Use *implicit links* as described in [26][27] namely name tracing, and relationships given by documentation structure. In particular, all representation relationships can be established through name tracing in such a way that all documentation entities related to the same logical entity can be identified by the same name. All refinement relationships can be established through documentation hierarchy. That means that the documentation entity corresponding to the logical child entity is part of the documentation entity corresponding to the logical parent entity. If manual link setting should be avoided altogether (e.g., because of missing tool support), dependency relationships can be established through referencing and name tracing. Related documentation entities reference each others´ names. In addition, all the instances of documentation entities are tagged with a “d”, if this instance describes a logical entity, and tagged with an “r”, if this instance only references this entity. Thus, in case of name tracing the search for the related entity is implemented through name search. Using the tags, all references and descriptions can be found, while accidental usages of these names (e.g., just for explanation) are ignored.

6.) Provide guidelines for searching and copying recycling candidates that make use of the explicit and implicit links. In particular, the guidelines should ensure that all documentation entities related to a recycling candidate are copied together with the candidate.

5.4. Experience

The activities above are illustrated in the following by our results and experience.

From the change scenarios described in Section 5 and the domain know-how provided by DC, Fh IESE identified the set of recycling candidates for the RD published in [28]. The majority of recycling candidates deal with “functions”, “environmental items” and “devices”. But product aspects, such as “country”, or hardware aspects, such as “voltage”, are also likely to be recycled.

The analysis of the conceptual models resulted in several opportunities for restructuring: As an example for the reduction of representation relationships consider Figure 1 once more. It shows the documentation types of the original document. Besides the “Title of function” and “Description of function” within the complete “Functional description”, entity types in the “Product overview” also represented a “complex function”, namely the “Picture”, “Title of short overview“, “Title of function overview”, and “Function overview description”[1]. The “Short function overview” characterized the “function” in one sentence; the “Function overview” described the “function“ in one paragraph. Both only served to give a short overview. Thus, we replaced the “Short function overview” with the “Function overview” and saved one representation relationship. Figure 2 shows the part of the template corresponding to the restructured version of Figure 1. However, typically at least one representation relationship remains, because major entities will be mentioned in the introduction as well as in the functional description. This redundancy is essential for the understandability of the document.

```
1.Product Overview
1.1   Function overview
      1.1.1 Function name1
      …..
      1.1.2 Function name2
```

Figure 2: Template restructures original document

Figure 3 gives another example for the reduction of representation relationships. In the original document, each “Input item” included in an “Input list” of a “Function” had a representation relationship to the “Pin” and the “Input signal description” (namely, through naming the input “S1.T_OPEN”). The reference to the pin number, and thus the first representation relationship can be omitted,

[1] Note that for the sake of clarity, not all relationships are shown in the figure.

since the pin number can be inferred through the refinement relationship between “Input signal description” and “Pin description”.

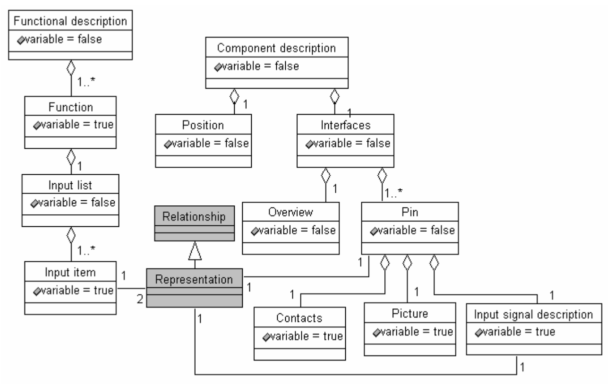

Figure 3: Representation relationships of “input item”

Figure 4 shows another part of the template. We focused especially on the description of the functions, since they are the major recycling candidates. One obvious choice for function description are automata or corresponding tabular representations.

7. Function
7.1. Function Fname1
7.1.1 Input list
Iname1:…
Iname2:…
7.1.2 Output list
Oname1:..
Oname2:..
7.1.3.Behavior description
precondition:…
input :..
output :…
interrupt:…
failure:…
timeout:…

Figure 4: Template refines original document

However, the description of the ECU functions is not detailed enough to provide all information for automata descriptions. Since we did not want to enlarge the documents, we identified six major information categories described in the documents, namely precondition, inputs and outputs of functions as well as conditions for interrupt, and timeout and fault handling.

The link setting guidelines use the following principles to reduce explicit links: The recycling candidates of type “item” and “function” have to be given a unique name, which has to be used consistently throughout the document. This ensures that representation relationships need not be set explicitly. Related entities are identified through searching for the specific name. Refinement relationships are realized through the documentation hierarchy. That means that, for example, in the section “functional description”, each “atomic function” that refines a “complex function” has to be described as part of the “description” of the “complex function”. If an “atomic function” is part of several “complex functions”, only a “title” is included in all complex functions “description”, and the detailed description is given as a separate “function” in the “functional description”. Similarly, dependency relationships are realized through documentation hierarchy and name references, or through explicit links. In our experience, explicit links are, of course, more comfortable. If a document is reused more than once, then at first, name search should be used. While navigating through the document, links should be set explicitly to be used in later recycling steps.

The guidelines for searching and copying are straightforward. If a documentation entity (e.g., a function “Seat Control”) is to be recycled, all documentation entities related through representation and refinement relationships (i.e., including the same name and all subparts of the corresponding “Description”) have to be copied as well. Tracing the dependency links leads to further documentation entities that possibly have to be copied. Of course, this procedure has to be applied recursively.

Our solution is able to handle all change scenarios given by DC (see Section 5.1). In the following, we discuss one example. It shows how our solution helps to avoid omissions and inconsistencies when copying a function and related items. Suppose that all except the function “Seat Control” should be recycled. The developer copies all function descriptions except the one for “Seat Control”. When copying the “Input signal description” and “Output signal description”, s/he checks for each of the inputs or outputs of “Seat Control” whether it is on the “Input list” or “Output list” of another function. If it is, it can be copied without change. If not, the corresponding description is not copied.

Of course, there are more complicated recycling steps that involve adaptation. Suppose, for example, that the input “Door_open” is to be recycled, but its type changes from “input signal” to “CAN-signal”. Because the explicit representation relationship between the “Input item” of a “Function” and the “Pin” was removed (see Figure 3), all “Function descriptions” can be copied without change. However, because “Door_open” changes its signal type from “PinSignal” to “CANSignal”, the “CAN-description” instead of the “Input signal description” has to be instantiated.

More detailed guidelines for the instantiations (giving technical details on how to adapt a documentation entity depending on the content) require further research.

Other constraints and risks, as well as benefits of our approach are discussed in the following chapter.

6. Benefits and Risks

The process helps the developers to reduce effort for recycling because the template localizes as much information as possible. It also helps to prevent omissions and inconsistencies, because it supports copying a documentation entity together with its related entities.

The process does not require any change in tool support, because it can be implemented with the search feature of an editor, if all links are realized through names and references. The additional effort required from the developers is the effort to use consistent naming, to adhere to the template and make relationships explicit. Because we only put information in the template that was in common with other RD, we did not enlarge the documents.

The major risk of our approach is that developers are not disciplined and do not adhere to the template and the guidelines. This would destroy the implicit naming links and the structure. This risk has to be mitigated through organizational measures.

Another risk are omissions and inconsistencies due to semantic dependencies (e.g., between the timeout behavior of the two functions). There are two major sources of semantic dependencies: relationships through a common usage context or because of design decisions. The former can be identified through capturing the usage context in terms of use cases for ECU (see [29]). The latter is supported by capturing additional information about design rationale for the requirements (see [30]). Both would at first significantly enlarge the effort (because of the additional effort for capturing), but would pay off in the long run. It is, however, an open question whether this effort would be more worthwhile than the effort for a full product-line approach. Our approach does not include these additions so far. However, the CSM and CDM can easily be extended to cover the identified semantic dependencies.

There does not seem to be an alternative simpler than our approach. The traceability guidelines without the template are not sufficient, because without the template the number of relationships cannot be minimized. The template without the traceability guidelines is not sufficient, because the template alone does not prevent omissions and inconsistencies in case of selective recycling. Another approach for minimizing manual link setting is automatic trace capture as described in [31]. This approach was not viable because of the additional tool support required.

Creating the CSM and CDM and deriving a template and guidelines requires effort, but this effort must be spent only once. The CSM can also be used for other applications (e.g., motor control, seat control) or types of embedded systems, such as building automation. The CDM can easily be adapted for more structured natural language documents, like use cases or tables. Furthermore, because of the well-defined relationships, the analysis of the conceptual models can be partially supported by a tool.

People responsible for the development process benefit from our approach. The whole approach can be seen as a first step towards product lines, where the template gathers the information on commonalities, not on variabilities. If our approach is used for some time, new insights will be gained on how new RDs differ. This can be used to refine the template. In addition, one can exploit our approach to support other traceability goals.

7. Main Contribution and Future Work

We described a solution to the recycling problem based on abstraction and traceability. In addition, we described our process to develop this solution based on a CSM and a CDM. This process was already validated in the domain of building automation. [22] shows how to derive guidelines for changing requirements and design documents based on a CSM and a CDM for use cases and UML-models. Experimental results showed a significantly beneficial influence of the guidelines on the correctness and completeness of a predicted set of change impacts in comparison to traditional development guidelines. Thus, we are convinced that our approach pays off for this domain and goal as well. Furthermore, we are sure that the process can be tailored to further domains and traceability goals, such as change and project management.

We see two directions for future work for the recycling problem:

- More fine-grained recycling would be supported through the provision of default values and default sentences in the template.
- Better understanding of the overall purpose of recycling candidates could be achieved through capturing the context and rationale.

8. Acknowledgements

We thank Joachim Weisbrod for helpful feedback during the literature study and the development of our approach. In addition, thanks goes to Erik Kamsties, Kirstin Kohler, Daniel Kerkow, and the anonymous reviewers for helpful comments. This work was partially funded by the German BMBF in the project QUASAR under grant VFB0004A.

References

[1] T. Biggerstaff, A. Perlis (Ed.) “Software Reusability. Volume 1 Concepts and Models”, *ACM Press Frontier Series*, 1989.

[2] W. Lam, “A case-study of requirements reuse through product families”, *Annals of Software Engineering,* 5 , pp. 253-277, 1998.

[3] M. Fowler, “Analysis Patterns”, *Addison Wesley*, 1996.

[4] W. Lam, S. Jones, and C. Britton, “Technology Transfer for Reuse: A Management Model and Process Improvement Framework”. *ICRE’98*, pp. 233- 240, 1998.

[5] L. Cybulski and K. Reed, “Requirements Classification and Reuse: Crossing Domain Boundaries”. W. B. Frakes (Ed.): *ICSR-6*, LNCS 1844, pp. 190-210, 2000.

[6] A. Sutcliffe and N. Maiden, “Domain Modelling for Reuse”, *ICSR*, 1994.

[7] S.R. Faulk, “Product-Line Requirements Specification (PRS): an Approach and Case Study”, *RE’01* , pp. 48-55, 2001

[8] F. Kiedaisch, M. Pohl, J. Weisbrod, S. Bauer and S. Ortmann, “Requirements Archaeology: From Unstructured Information to High Quality Specifications”, *RE’01*, 2001.

[9] F. Kiedaisch, M. Pohl, J. Weisbrod, S. Bauer and S. Ortmann, “Experiences on Outsourcing Requirements Specifications”, *EuroSPI’01*, 2001.

[10] W. Lam, J.A. McDermid and A.J. Vickers, “Ten Steps Towards Systematic Requirements Reuse”, *Requirements Engineering Journal*, No. 2, pp. 102-113, 1997

[11] A. Sutcliff and N. Maiden, “The Domain Theory for Requirements Engineering”, IEEE Transaction on Software Engineering, vol. 24, no.3, March 1998

[12] B. Ramesh and Edwards, E., “Issues in the Development of a Requirements Model”, ISRE’93, pp. 256-259, 1993

[13] K. Pohl, “Process-Centered Requirements Engineering”. WS, 2nd Edition, 1996.

[14] A. von Knethen and B. Paech, “A Survey on Tracing Approaches in Practice and Research”, *IESE-Report No. 095.01/E*, January 2002.

[15] B. Ramesh, T. Powers, C. Stubbs, and M. Edwards, “Implementing Requirements Traceability: A Case Study”, *ISRE*’95, pp. 89-95, 1995.

[16] M. Lindvall and K. Sandahl, “Practical Implications of Traceability”, *Software – Practice and Experience*, Vol. 26, No. 10, pp. 1161-1180, 1996.

[17] O. Gotel and A. Finkelstein, “An Analysis of the Requirements Traceability Problem”, *ICRE’94*, pp. 94-101, 1994.

[18] B. Ramesh, “Factors influencing requirements traceability” *Communications of the ACM*, Vol. 41, No. 12, pp. 37-44, 1998.

[19] B. Ramesh and M. Jarke., “Towards Reference Models for Requirements Traceability”. *IEEE Transactions on Software Engineering*, Vol. 27, No. 1, 2001.

[20] http://www.telelogic.com/

[21] J. Bedocs. “A Data Architecture for DOORS Projects”, InDOORS 1999, http://www.telelogic.com/

[22] A. von Knethen, “Change-Oriented Requirements Traceability. Support for Evolution of Embedded Systems”. *PhD Theses in Experimental Software Engineering*, Vol. 9, Fraunhofer IRB, 2002.

[23] A. von Knethen, “A Trace Model for System Requirements Changes on Embedded Systems”. *IWPSE’ 01*, 2001.

[24] D. Parnas and J. Madey,. “Functional Documentation for Computer Systems Engineering”. *CRL Report 237*, McMaster University, Hamilton, Ontario, Canada 1991

[25] R. Bharadwaj and C. Heitmeyer, “Hardware/Software Co-Design and Co-Validation: Using the SCR Method”, *HLDVT’99*, Nov. 1999.

[26] M. Lindvall, “A Study of Traceability in Object-Oriented Systems Development. *Licentiate thesis Linköping Studies in Science and Technology No 462*, Linköping University, Institute of Technology, Sweden, 1994.

[27] D. Leffingwell and D. Widrig, “Managing Software Requirements. A Unified Approach”, *Addison-Wesley*, 2000.

[28] F. Houdek and B. Paech, “Das Türsteuergerät – eine Beispielspezifikation”, IESE-Report, 002.02/D, 2002

[29] I. Alexander and F. Kiedaisch, “Towards Recyclable System Requirements”, ECBS’2002, p.9-16, 2002

[30] A. Dutoit and B. Paech, “Rationale Management in Software Engineering”, *Handbook of Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, 2001.

[31] K. Pohl, “PRO-ART: A Process Centered Requirements Engineering Environment”, M. Jarke, C. Rolland, A. Sutcliffe, R. Dömges (Ed.) “The Nature of Requirements Engineering”, *Berichte aus der Informatik, Shaker Verlag*, 1999.