update of AIB No. 91–16, to appear in Proceedings of COMPSAC '91

Bridging the gap between Requirements Engineering and Design

Jürgen Börstler and Thorsten Janning

Lehrstuhl für Informatik III,
Aachen University of Technology, W–5100 Aachen, Germany

Abstract

Modern CASE tools raise the need for traceability, i.e. the ability to control the consistency between software documents produced at different stages of the software life–cycle. There have been several approaches to generate a design from a given requirements definition starting with Structured Analysis/Structured Design up to the recent Object Oriented Analysis/Object Oriented Design discussion. Most of these solutions have the disadvantage of being unidirectional and/or batch–oriented. They do not allow for the propagation of incremental changes from the requirements definition to the design or vice versa. In this paper we describe the transformation between an integrated requirements engineering language based on Structured Analysis and the Entity Relationship Model on one side and a modern design language on the other side. The transformation works incrementally and is sensitive to changes to already transformed parts. We highlight the potential but also the limits of this approach.

1 Introduction

The first phases of the software lifecycle, i.e. requirements engineering and design, have been accepted as being very important activities to improve the quality of software products and to reduce the costs for their development. During requirements engineering the problem space is investigated and described by the requirements definition. The design is the first important step into the solution space. Since both are very different views of the same software system different languages were developed for both phases.

It is obvious that some parts of a design directly depend on related parts of the requirements definition. Therefore, it seems to be suggestive to use the requirements definition as a starting point for deriving at least an initial design. For this reason, algorithms transforming a requirements definition into a design are as old as requirement engineering languages ([17]).

But there are very strict limits to such a transformation mechanism. Of course, only parts of the complete design can be generated. It is also difficult to incorporate design knowledge and experience into the transformation process, i.e. the deeper the designer is involved into the transformation process the better may be the resulting design. And, last but not least, the quality of a generated design can certainly not exceed the quality of the underlying requirements definition.

The success of a transformation mechanism also heavily depends on the languages used for describing the source and the destination of the transformation. For this reason, we first describe the features of our requirements engineering and design languages on the basis of an example system. The third chapter outlines the transformation algorithm. Chapter four compares our approach to those found in the literature. Chapter five briefly describes a tool which implements the proposed algorithm and supports the proposed traceability. The last chapter gives a summary and sketches some future work.

2 The underlying languages

In our languages we have integrated elements proved to be useful in the most widespread languages. To introduce both languages we use a simple project management system managing several software projects and a staff of team members. Team members have to be declared responsible or not responsible for parts of one or more projects. Furthermore, the costs of the projects resulting from the costs caused by the team members have to be computed. The same system is then used to demonstrate the transformation process.

2.1 The requirements engineering language

2.1.1 The functional view: To describe the functional view of the requirements of a software system Structured Analysis (SA, [6]) has proved to be very useful. Hence, we decided to take a dialect of it for our functional part of the requirements engineering language. Since SA is well known we only describe one special feature of our SA dialect which is necessary to understand the transformation described in this paper.

We distinguish two classes of data stores. The data stores in the upper levels of our data flow diagram (DFD) hierarchy usually are artificial unifications of data stores in the next lower level in the DFD hierarchy. We call these data stores unification data stores. In our example ProjectDataBase is such a unification data store (see figure 1 (a)).

On the other side there are collections of objects of the same type which are called atomic data stores. They usually appear in the lower levels of the DFD hierarchy. Examples for atomic data stores are Staff, Projects or Responsibilities (see figure 1 (b)).

Atomic data stores have to be related to the types of the entries in the data store. These types correspond to an entity type or a relationship type defined in the data oriented view. For example the data store Staff is declared as a collection of instances of the entity type TeamMember (see figure 2 (a)).

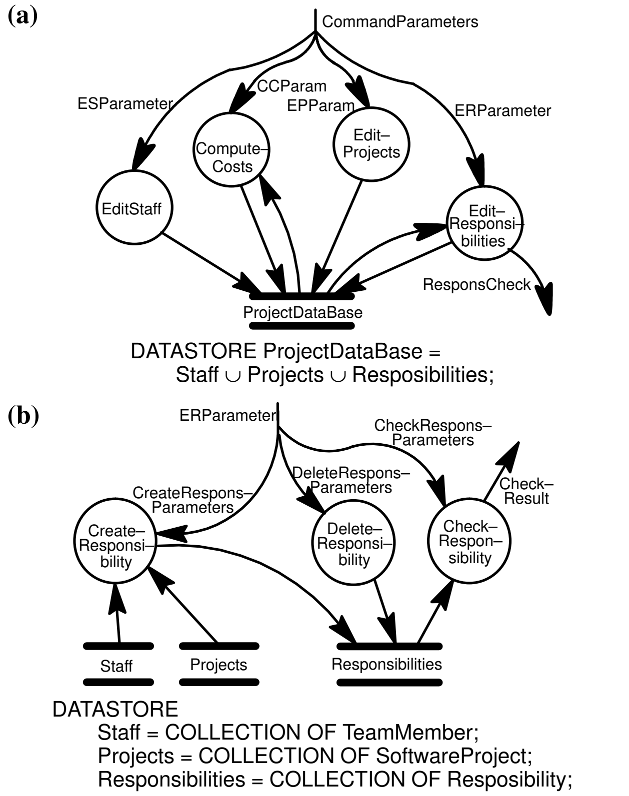

Figure 1: DFD "ProjectManagement" ((a)) and the refining DFD "EditResponsibilities" ((b)).

2.1.2 The data oriented view: For the data oriented view of the requirements many authors ([13],[16]) propose the entity relationship (ER) model as an useful extension to SA because pure SA is not sufficient enough to describe complex data structures. In our requirements engineering language an extended ER model plays the role of the data dictionary of pure SA. Again, we only mention those extensions to the ER model which are necessary to understand our transformation method.

The first extension is the generalization. It introduces an inheritance relationship between a supertype and one or more subtypes. That means that all attributes defined for the supertype are also defined for the subtypes. The subtypes may add new attributes. In our example the subtypes Employee and SelfEmployed inherit the attribute structure of the supertype TeamMember (see figure 2a).

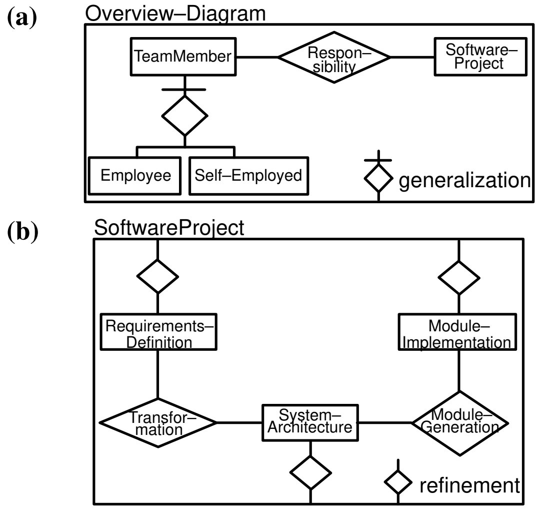

Figure 2: ER diagram (overview diagram) with generalization ((a)) and refinement ((b)).

Furthermore, entity types and relationship types having a complex structure may be refined by a whole entity relationship diagram (ERD). Such complex types are called molecule types, the types in the refining ERD are called component types. In our example in figure 2b we have the molecule type SoftwareProject and the component types RequirementsDefinition, ModuleImplementation, etc. The relationship between molecule type and component type has the semantic of "consists of". It may be specified by the cardinalities 1:n or 1:1. In the first case the component type is called a multiple component type, in the second case a single component type. By doing this we get a hierarchy of ERDs in our data oriented view similar to the DFD hierarchy in the functional view. The top–level ERD is called overview diagram (see figure 2a).

2.2 The design language

It is widely agreed that the quality of a design document is essentially induced by (data) abstraction, information hiding, encapsulation, structuring, coupling, and cohesion ([3],[12]). It is therefore important to define a design language that covers concepts like (different kinds of) modules, subsystems, parameterization of modules and subsystems, inheritance, usability, and locality.

There are further concepts like versions, configurations, concurrency, and persistence, which are important. These further concepts are only partly realized in our design language. Since the transformation algorithm does not support these concepts at all, we omit their description.

2.2.1 Modules: Our design language supports four kinds of modules to encapsulate data types, data objects, functions, and subsystems.

A data type module encapsulates an abstract data type similar to the class concept known from object–oriented programming languages ([11]). It is used as a pattern for the dynamic creation of objects of the same type, which can only be created and manipulated by the operations offered at the export interface of the data type module.

A data object module encapsulates a single, static data structure, which can not be created or deleted from outside the module. It can be seen as a named instance of an abstract data type. It is useful to make important objects visible in the architecture, for example collections of objects, which are used by several different modules.

A function module encapsulates one or more functions, which do not manage an own memory. Its purpose is to designate modules which transform input data to output data independent of an internal state.

A subsystem encapsulates a set of interrelated modules, which are local to this subsystem. A subsystem has its own interface and therefore hides its internal structure. In this way the subsystem is a consequent continuation of abstraction and information hiding on architecture level.

2.2.2 Parameterization: Modules can be parameterized. Parameterization can be constrained or unconstrained. The formal parameters are described by a type name in case of unconstrained parameterization, or by referring a data type module, which encapsulates the constraints by the resources in its export interface.

2.2.3 Relationships between modules: Modules may use resources offered in the export interfaces of other modules. We therefore introduced the usability relation.

Besides the usability relation we introduced the inheritance relation, which is only permitted between data type modules. The inheritance relation imposes a supertype–subtype hierarchy on top of the data type modules. This ensures that the export interface of every subtype at least contains the same resources as the interfaces of their supertypes.

Name conflicts, which can occur in consequence of multiple inheritance, must be resolved by the user.

Data type modules can be abstract, that means they may export resources, which defer their implementations to subtypes.

2.2.4 Usage of subsystems: Subsystems can be used to encapsulate subarchitectures and, therefore, are serving as patterns for typical situations. Examples for such situations are:

- The realization of collections of entries of some type (see figure 3)
- The realization of complex functions, which are realized by several subfunctions, whereby these subfunctions are exclusively used to realize the complex function (see figure 4)

Figure 3 shows the graphical representation of the first situation. The subsystem is composed of the modules Collection and Entry. The data type module Entry encapsulates the type whose instances are collected in the data object module Collection. Therefore Collection must be connected to Entry by an usability edge. Both modules contribute to the interface of the subsystem. We denote this by double arrows.

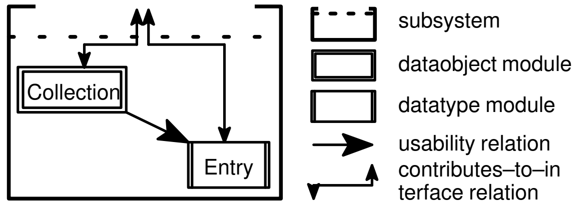

Figure 3: Entry–collection subsystem

In fact, both modules can be very complex and therefore be subsystems themselves. It is also possible to define Collection and/or Entry as generic parameters to the subsystem.

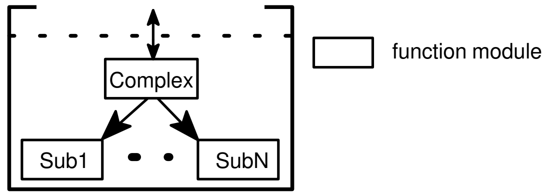

Figure 4: Complex–function subsystem

Figure 4 shows the graphical representation of the second type of subsystem. It is composed of functional modules, whereby only the module encapsulating the complex function (Complex) contributes to the interface of the subsystem. This module has an usability edge to each module encapsulating a subfunction (Sub1, ..., SubN).

If Sub1 to SubN are very complex they may be subsystems themselves. It may also happen that there are common subfunctions. In this case all modules using this common subfunction must be connected to the encapsulating module by an usability edge.

3 The transformation algorithm

To cope with the complexity of the transformation process we divided it into several steps:

One of the most characteristic features of modern design languages is that they enable designers to describe a module architecture in defining certain layers of (data) abstraction. In our requirements engineering model we have such a data abstraction hierarchy. It is described in the information model with its hierarchy of ERDs.

In a **first step** we take this part of the requirements definition to generate a skeleton of data abstraction modules with some rough data abstraction layers in a module architecture. The allocation of resources for these modules is shifted to the last step.

In a **second step** we will take into account the data stores of the functional model to extend this hierarchy of data abstraction modules.

In the **last step** we will use the processes and data flows of the functional model to allocate resources for existing data abstraction modules or to generate function modules.

3.1 The transformation of the data oriented view

In our requirements engineering language an entity type describes the common structure of some entities. In our design language types are encapsulated in data type modules together with the resources to change instances of the type. Thus, our first transformation rule is:

- ♦ Each entity type is transformed to a data type module exporting an abstract data type.

A relationship type is the type description of the tuple of the participating entity types. Besides introducing relations between entity types relationship types may also carry additional information defined as attributes. Therefore they should be represented by their own modules on design level. We can define the following transformation rule:

- ♦ Each relationship type is transformed to a data type module exporting an abstract data type.

Until now data type modules are not connected with each other. But the supertype and subtypes of a generalization are related heavily with each other. We can use the inheritance relationship between modules for the next transformation rule:

- ♦ Inheritance relationships are drawn from the data type modules corresponding to the subtypes of a generalization to the data type modules corresponding to the supertype of the generalization.

The transformation of the overview diagram of our example (see figure 2a) by the first transformation rules is shown in figure 5.

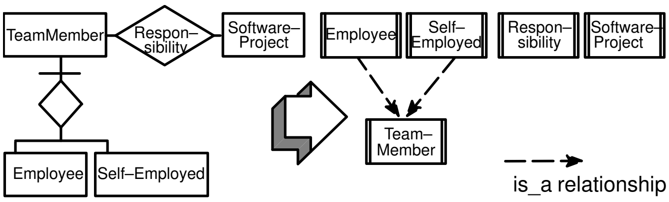

Figure 5: Transformation of the overview diagram.

The hierarchy of the ERDs of our information model describes a data abstraction hierarchy. Every time a complex type is refined a new abstraction level is created. In the design the data abstraction hierarchy is represented by the usability relationship between modules of different abstraction layers, therefore the next transformation rule is:

- ♦ Usability edges are drawn from the data type module of a molecule type to the data type modules of its component types.

For our example we see in figure 6 how the refining ERD of SoftwareProject is transformed according to our transformation rule.

Transforming the information model in this way we get an acyclic graph of data type modules. The upper level of this graph, i.e. those modules not being used (until now) are the modules encapsulating the entity types or relationship types of the overview diagram.

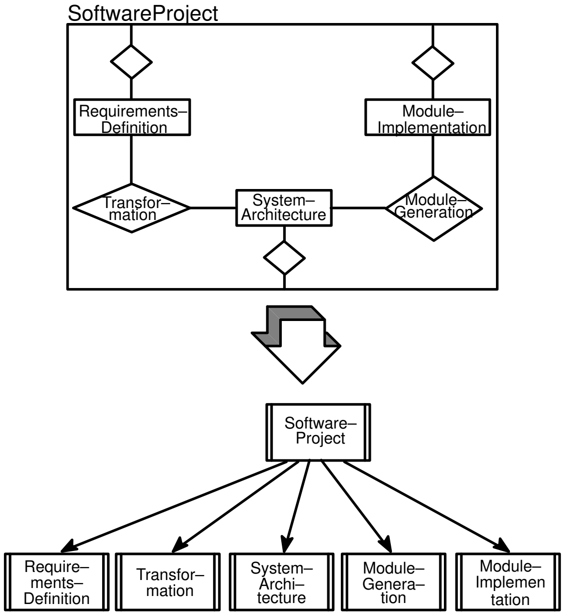

Figure 6: Transformation of a refining ERD.

3.2 The transformation of the functional view

3.2.1 Data Stores: Data stores are the connecting link between data oriented view and functional view within the requirements definition. Therefore, they have to be the first part of the functional view being transformed. However, we do not have to transform all data stores into such collection modules.

Only atomic data stores are relevant for our transformation. This is because unification data stores only represent sets of atomic data stores in higher level DFDs. As a consequence, unification data stores do not contain any additional information and do not have to be transformed.

The collection module corresponding to the data store together with the data type module corresponding to the entry type is an example for the first subsystem situation of the last chapter, the entry–collection subsystem. Therefore, in our example the modules Staff and TeamMember have to become part of a subsystem StaffSystem.

According to this, we can define the next transformation rules:

- ♦ Each atomic data store related to an entity type or a relationship type is transformed to a data object module being a collection of instances of the entry type. A usability relationship has to be inserted from this data object module to the data type module corresponding to the entry type. Unification data stores must not be transformed.
- ♦ The data object module corresponding to the data store and the data type module corresponding to the entry type must become components of a common generated subsystem. If the entry type is a molecule type, all data type modules corresponding to component types must also be

incorporated into this subsystem.

Having transformed all data stores according to those rules the result for our example can be seen in figure 7.

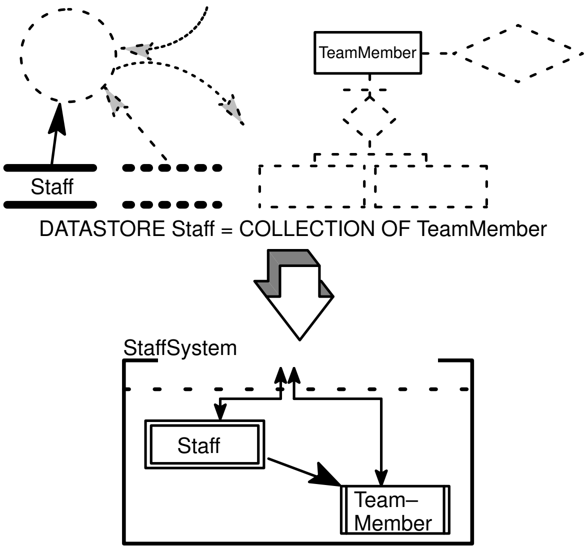

Figure 7: Transforming an entity type together with a related data store into an entry–collection subsystem.

3.2.2 Processes and data flows: In a last step we have to transform the processes and data flows of the DFDs. It is one of the characteristics of SA that we cannot distinguish those processes syntactically which are typical resources of data abstraction modules from those processes which have the character of function modules or resources of them. Therefore, we will present three alternative transformation rules for processes. The designer will have to decide which rule is the most appropriate in each case.

We mentioned that there is a hierarchy of DFDs. On the other hand we want to get a hierarchy of modules in our design language. If we want to study the transformation of processes we have to imagine that we will study the transformation of the DFD hierarchy implicitly. It seems to be natural to transform the DFD hierarchy directly into a module hierarchy (cf. transition from SA to SD, [13]) according to a rule like:

- ♦ Generate a function module exporting one procedure with the same name for every data flow diagram.

In our example the processes ComputeCosts, EditStaff, EditProjects and EditResponsibilities will be transformed according to that rule which will be called the horizontal transformation rule. We see the application of the transformation of those processes in figure 8. Here, the designer also has decided to create a complex–function subsystem (cf. figure 4).

However, applying this rule to each process, i.e. also to the atomic processes, we would get a very large number of function modules of very small size. Therefore, we offer two other transformation rules for atomic processes to enable the designer to pack several processes into larger design objects. We will call them upward transformation rule and downward transformation rule.

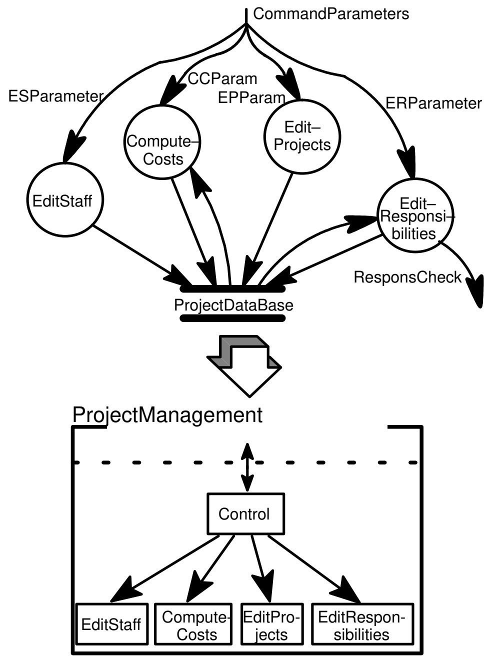

Figure 8: Applying the horizontal transformation rule and generating a complex–function subsystem.

We want to explain the upward transformation rule using an example: Within the DFD EditResponsibilities we have three atomic processes each of which would become a function module if they were transformed according to the horizontal rule. We think those modules would be much too small. What this DFD obviously expresses is that to edit a responsibility means either to create or to delete or to check it. It expresses something like an interface of the function module EditResponsibilities. So, the following transformation rule, the upward rule, would match the situation much better:

- ♦ Generate a procedure declaration for each atomic process within the function module corresponding to its parent process.

The procedures of the example above even have to be exported at the interface of the function module. However, we can easily find an example where the procedures only have to be used in the body of the module and, therefore, should not be exported. The DFD ComputeCosts obviously expresses how to export the resource ComputeCosts. It has the character of an implementation guideline. So, we can formulate the upward transformation rule more precisely:

- ♦ Let a DFD contain processes which have been transformed according to the upward rule. If the designer identifies the DFD as one that expresses an interface of a function module he has to decide that also procedure exports must be generated. If he identifies the DFD as one that expresses an implementation of a module he must not generate exports of the procedures.

The application of the downward rule depends on the granularity of the requirements definition. We have seen that the process CreateResponsibility should become a procedure at the export interface of the module EditResponsibilities. The semantic of such a process would be the change of the database but also the representation of the new state at the user interface. So, it is more than a resource of the abstract data type module Responsibility also having the name CreateResponsibility. However, it is possible that a requirements engineer wants to build a model with just that process which only changes the database.

Although we think that such a requirements definition model would be too detailed we cannot and do not want to forbid such a use of our requirements engineering language. So, we need a third transformation rule for process declarations, the downward rule:

- Let a process be connected to a data store by a data flow. Then, this process has to be transformed to a procedure export at the interface of one of the data abstraction modules encapsulating the data store or the entry type.

4 Relation to other work

There are several approaches in the literature to a transformation as described before.

These approaches can be divided in three groups: The structured approaches, the object oriented approaches, and the combined approaches.

In the structured approaches the main attention is paid to functional decomposition. Their approaches describe the transition from a requirements definition stated in SA to an initial design represented in structure charts (hierarchical procedural components) using transform and/or transaction analysis ([6],[13],[17]). This design is then analyzed with respect to coupling and cohesion of the components. The components are then reorganized to improve coupling and cohesion ([10]). In a final packaging phase the components are then grouped to executable programs and load units. All of these approaches fall short in data modelling, because their attention is focused on the functionality of a system.

In the (pure) object oriented approaches there is no formal description for the requirements. Object Oriented Analysis (OOA, [4]) is focused with the identification of objects in the problem domain and assigning behavior to these objects. After that the objects are grouped into classes of similar objects and relationships are installed between these classes. These classes and relationships can then simply be mapped into the constructs of object oriented programming languages. These approaches fall short, because they neglect the fact that the real world is not solely modelled with objects. Another disadvantage is the informality of OOA. Therefore it is not possible to formulate a transformation algorithm. However, in restricted domains for e.g. data base design (extended) ER models have proved to be very useful for the requirements engineering phase ([5]).

Since the structured methods have proved to be useful for requirements engineering, and object oriented methods fit well together with data modelling it seems reasonable to use combined approaches.

Several authors propose an approach similar to ours. They combine SA and ERDs ([16]) to an object oriented requirements engineering method. The consistency rules inside their requirements engineering language are much more course–grained than ours. Therefore, it is not possible to build sophisticated transformation tools which allow for the propagation of incremental changes between requirements definition and design.

Other approaches try to find data abstractions analyzing the structure of the DFDs ([1],[2],[14],[15]). These methods tend to create very complex diagrams which are difficult to understand or the transformation process is not well described. There are also some authors which totally reject (simple) SA as a front–end for object oriented design ([7]). It is our opinion too that a useful design can not be derived from a requirements definition which has not considered data modelling from its very beginning.

5 The transformation tool

During the evolution of both languages we recognized that a transformation as described in this paper would be very useful to generate an initial design, which is consistent with the requirements definition. This consistency can be checked by means of a separate transformation document (see figure 8). The transformation document is used to record all informations to support traceability, like source and target increment, the rule which was applied, and an internal status.

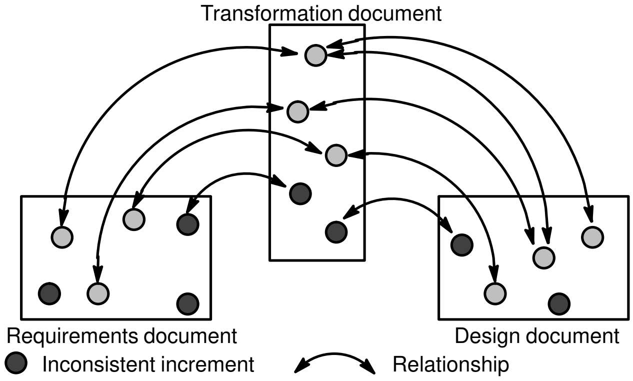

Figure 9: Interrelationships between requirements and design

Practical experiences showed that such a tool should support incremental and interactive transformations, because requirements tend to change very often. In addition it is also very useful to be able to lookup those parts of the requirements document which correspond to the design increments currently under work and vice versa. Therefore, the transformation is controlled by an incrementally working consistency checker which is based on the transformation document. In this way we reached a high integration between the requirements definition document and the design document. This allows us to trace the effects of each change immediately, e.g. to check all increments which are not transformed yet. A more detailed discussion on this subject can be found elsewhere ([8]).

Although heavily interrelated the requirements definition and the module architecture contain different information. A proposal for an architecture frame can not be created without some knowledge about useful design decisions. Thus, it only seems to be possible to offer a semiautomatic transformation tool ([9]).

Our transformation tool proposes the horizontal rule for DFDs and the upward rule for atomic processes only as defaults. However, the user can correct this and decide that he prefers to apply either the horizontal, the upward, or the downward rule if the constraints (for example that the process is connected with the data store by a data flow) are fulfilled. With this alternative application of all three rules the designer is involved very deeply into the transformation process. On the other hand he is not weakened during the transformation by hundreds of questions because there is a unique default transformation.

It should be noted that we also support the creation of traceable relationships by hand. This is very useful to support exceptional situations.

6 Conclusion and future work

We described an incremental and interactive transformation from a requirements definition into a design, which can be useful in several ways:

- Freeing the designer(s) from most of the burden of creating an initial design.
- Traceability of changes in the requirements definition and the design.
- Detecting requirements which are not considered in the design.

A lot of the design work must still be done independently from the transformation process, for example to detect further subsystems, make use of genericity, or to reuse parts of the system. Of course, some of this work may be shifted to the transformation process. For example the transformation tool can be extended to detect hierarchies of function modules as shown in figure 4 and package them into a subsystem. Another useful extension would be the integration of a reuse process into the transformation tool: To free designers from checking all modules generated by the transformation tool for their availability in the library of reusable components, we are thinking of ways to search for appropriate documents during the transformation process ([3]).

However, we think a lot of work can not be automated, because design is more than a compilation of the requirements definition of a system. Therefore, we want to support the manual installation of relationships between increments in both languages. This is necessary to relate requirements to parts in the design, which have not been generated by the transformation tool.

We know that both the requirements engineering language and the design language have to be extended by real–time features. We are just beginning to work in that area. Of course, also the transformation will be extended by those features. Nevertheless, we are sure that those extensions will fit with well together with our work described here.

We are also developing a metamodel for modelling languages which serves as a basis for the derivation of incremental transformers as described above ([8]).

References

[1] Alabiso, B. 'Transformation of Data Flow Analysis Models to Object–Oriented Design' *Proceedings of OOPSLA '88* (1988).

[2] Bailin, S. C. 'An Object–Oriented Requirements Specification Method' *Comm. of the ACM* Vol 32 No5 (May 1988).

[3] Börstler, J. 'Integrating Reuse into a Software Development Environment' *Proc. of the First International Workshop on Software Reusability*, Dortmund, Germany, (July 1991).

[4] Coad, P. and Yourdon, E. *Object–Oriented Analysis*, Yourdon–Press, New York, USA (1990).

[5] Kim, W. and Lochovsky, F. *Object–Oriented Concepts, Databases, and Applications*, ACM–Press, New York, USA (1989).

[6] DeMarco, T. *Structured Analysis and System Specification*, Yourdon–Press, New York, USA (1978).

[7] Firesmith, D. 'Structured Analysis and Object–Oriented Development are not Compatible' *ACM Ada Letters* Vol XI No 9 (Nov/Dec 1991).

[8] Janning, T. 'A Highly Integrated Environment for Analysis and Design' *submitted for publication* (1992).

[9] Janning, T. and Lefering, M. 'A transformation from requirements engineering into design – the method & the tool' *Proc. of the 3rd international workshop on Software engineering & its applications*, Toulouse, France (Dec 1990).

[10] Karimi, J. 'An Automated Software Design Methodology Using CAPO' *Journal of Management and Information Systems* Vol III No 5 (1987).

[11] Meyer, B. *Object–oriented Software Construction*, Prentice Hall, London, GB (1988).

[12] Nagl, M. *Methodical Programming in the Large – Modelling on Design Level* (in german, english edition in preparation), Springer, Berlin, FRG (1990).

[13] Page–Jones, M. *A Practical Guide To Structured Systems Design*, 2nd Edition, Yourdon Press, New York, USA (1988).

[14] Seidewitz, E. and Stark, M. 'Towards a General Object–Oriented Software Development Methodology' *ACM Ada Letters* Vol VII No 4 (1987).

[15] Ward P. T. 'How to Integrate Object Orientation with Structured Analysis' *IEEE Software* Vol No (March 1989).

[16] Yourdon, E. *Modern Structured Analysis*, Yourdon Press, New York, NJ, USA (1989).

[17] Yourdon, E. and Constantine, L. *Structured Design*, Yourdon Press, New York, USA (1978).