Resolving Requirement Conflicts through Non-Functional Decomposition

Eltjo R. Poort
LogicaCMG Arnhem, Meander 901, 6825 MH Arnhem, The Netherlands
Peter H.N. de With
LogicaCMG / Eindhoven Univ. of Technol., P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{eltjo.poort,peter.de.with}@logicacmg.com

Abstract

A lack of insight into the relationship between (non-) functional requirements and architectural solutions often leads to problems in real life projects. This paper presents a model that concentrates on the mapping of non-functional requirements onto functional requirements for architecture design. We build a framework that both provides a model and a repeatable method to transform conflicting requirements into a system decomposition. This paper presents the framework, and discusses two cases onto which the method is applied. In one case, the method is successfully used to reconstruct the high-level structure of a system from its requirements. The second case is one in which the method was actually used to create a system design fitting the stakeholders' needs, and that is reproducible from its requirements.

1. Introduction

The primary result of any architectural design process is a blueprint of a system, identifying the main components and their relationships from different views. A topic that is currently under close scrutiny is the derivation of these architectural components from the functional and non-functional system requirements. The well-known discipline of Functional Decomposition (FD) can be used as a basis, but will by itself rarely yield a system that fulfills the non-functional requirements. This is not surprising, since rules of Functional Decomposition only deal with generic best practices for achieving software quality, such as high cohesion and low coupling. FD has no rules to deal with system-specific quality requirements.

The need for an integrated approach for Functional Requirements, Non-Functional Requirements and Architecture was discussed in [12]. In [5] this subject is dealt with by first obtaining a functionality-based architecture, and then applying architectural transformations to satisfy the "Quality Requirements". A good example of a detailed method using this iterative approach is given in [7]. In this paper, we have chosen a more direct approach. Our aim is to develop a method for decomposing a system based on the conflicts in the system requirements. We have named this method Non-Functional Decomposition (NFD) to highlight the contrast with Functional Decomposition, and to emphasize the importance of Non-Functional Requirements in this process.

There are several other research groups working on the relationship between requirements and architecture of software. Publications of the Software Engineering Institute (mostly technical reports) show the development of a framework [2] and tooling [3] towards methodical architectural design. Another group has developed the Component-Bus-System-Property (CBSP) method for iterative architectural refinement of requirements [9, 10]. In [9], the need is mentioned to group artifacts to create an architecture, but no indication is given how to do this. NFD proposes a method for grouping and splitting of architectural entities based on requirements, and is complementary to the CBSP approach in that sense.

A clear benefit of the NFD approach is that it focuses on non-functional and other supplementary requirements right from the beginning, yielding a defined trace from those requirements to the system structure. Moreover, the development process and its requirements are also integrated in the approach, giving a better basis for architectural and project decision trade-offs.

The sequel of this paper is as follows. First, we will present and discuss some of the shortcomings of the generally accepted model for the architectural design process. We will then develop a refined model of this process. Then we will describe the process for deriving system structure from supplementary requirements that is based on this model. The succeeding sections then describe two cases: one in which the NFD method was used, and one in which it is applied retrospectively to show its validity. We conclude with a discussion.

Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)
0-7695-2172-X/04 $ 20.00 © 2004 IEEE

2. Motivation for Non-Functional Decomposition

Our interest for Non-Functional Decomposition is based on a number of distinct observations from our substantial experience in architectural design.

- *Cohesive force of supplementary requirements:* good architectures tend to cluster functions with similar supplementary requirements in the same subsystem.
- *Divide-and-conquer conflict resolution principle:* if a subsystem has to fulfill conflicting requirements, it is useful to separate the parts that cause the conflict(s).
- *Entanglement of function, structure and building process of software:* these three elements are highly interrelated.

NFD is a framework consisting of both a model of the elements involved in the architectural process, and a method for architecting software-intensive systems based on system requirements. It is a framework in the sense that it does not venture into the details of achieving specific quality attributes (or other supplementary requirements); there is ample literature available for each conceivable attribute. Rather, it highlights the relationships between these requirements, their conflicts and ways to resolve them. It also helps in making choices about the development process.

3. Model of requirements and architecture

3.1. Accepted model for architectural design

When studying the available literature on the relationship between system requirements and software architecture, the following widely accepted model emerges.

System Requirements are usually divided into Functional and Non-Functional Requirements. These Non-Functional Requirements (NFRs) are often referred to as Quality (Attribute) Requirements; these two are treated more or less as synonyms. A generally accepted principle is *the leading NFR principle*: in designing system architectures, the Non-Functional or Quality Requirements are at least as important as the Functional Requirements. In order to satisfy NFRs the software architect applies Architectural Strategies to the system design, such as design patterns, layering techniques, etc. The architect's task then becomes an n-dimensional optimization problem: find the combination of architectural strategies that yields a system with the best fit to the n NFRs.

The implicit model underlying this model is depicted in Figure 1. Although the simplicity of this view has its merits, in our experience it has some shortcomings. Particularly, the relationship between quality attributes and non-functional requirements is oversimplified, and it ignores the fact that

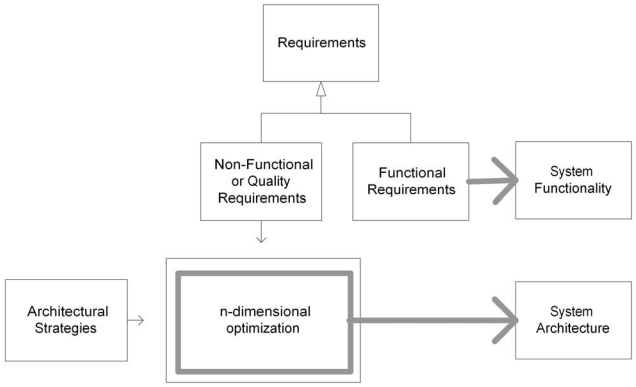

Figure 1. Accepted model of relationship between requirements and architecture.

functional requirements can be very important in system design. It also ignores that NFRs often put constraints on the system development process rather than on the system architecture, implying that architectural choices are not the only contributors to satisfy NFRs. Conversely, requirements on the development process like project deadlines and budget limitations can have a large impact on system architecture.

3.2. Refined requirements classification for NFD

Our new NFD model, as is illustrated in Figure 2, refines the classification of requirements, and is more detailed on the non-functional aspects. Two major differences come to the foreground: functional requirements are split into primary and secondary functional requirements, and the secondary functional requirements are grouped together with the non-functional requirements. This group is called *supplementary requirements*. Additionally, a distinction is made between two types of non-functional requirements: quality attributes and implementation requirements.

Let us now define the Primary and Supplementary Requirements groups in more detail.

Primary Functional Requirements are demands that require functions which directly contribute to the goal of the system, or yield direct value to its users. They represent the principal functionality of the system. The identification of primary requirements (which ones to select) is similar to determining which processes in an organization are primary processes. All primary requirements are functional (there are no non-functional primary requirements), but not all functional requirements are primary requirements, as will be explained in the next section.

Supplementary Requirements represent all other requirements imposed on the system. They can be functional or

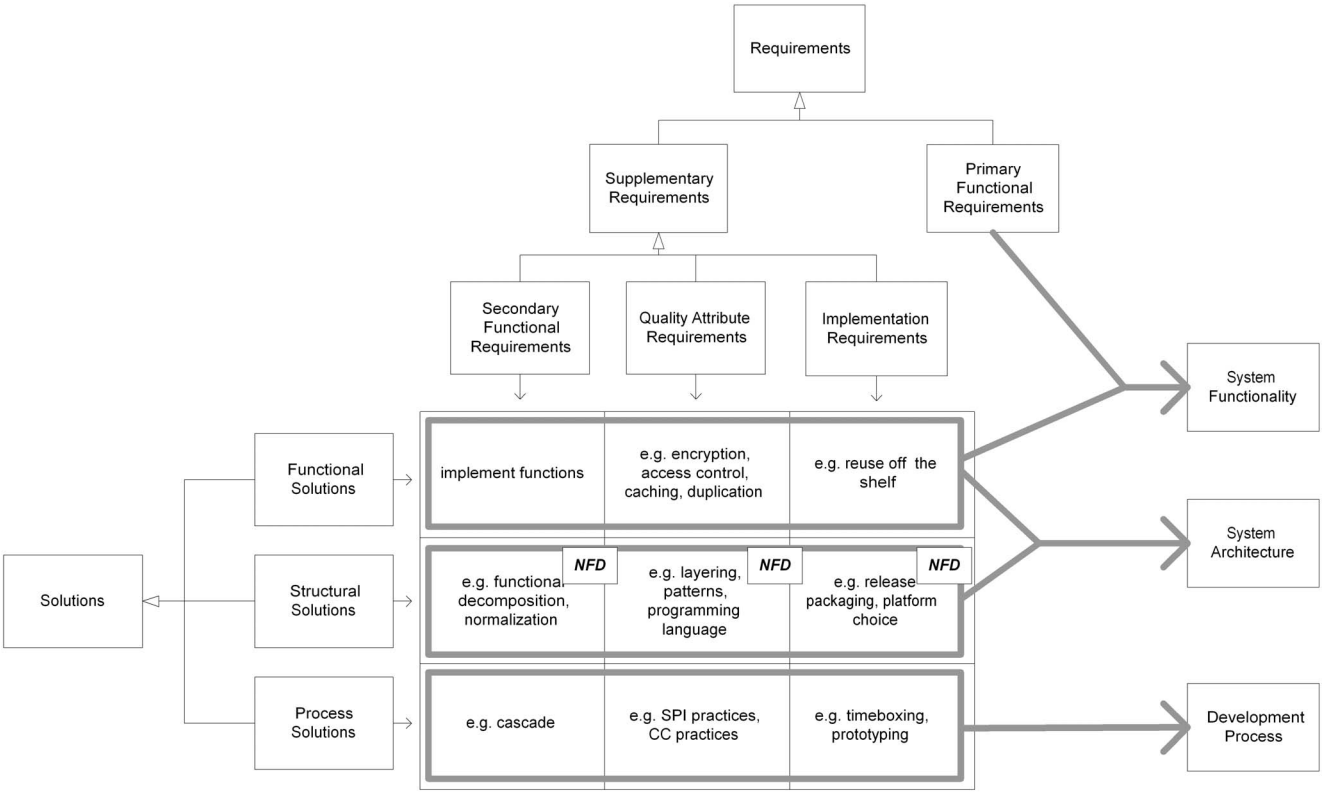

Figure 2. The NFD model of the relationship between system requirements and architecture.

non-functional. Supplementary requirements (SRs) are always *about* primary requirements, and usually put constraints on how the primary functionality is implemented. In the NFD model, the Supplementary Requirements are further divided into three subcategories:

1. *Secondary Functional Requirements* (SFRs) require functionality that is secondary to the goal of the system. Examples are functions needed to manage the system or its data, logging or tracing functions, or functions that implement some legal requirement. Like all other SRs, they usually apply to a particular subset of the primary requirements. For example, "All transactions in module X should be logged", or "access to data in chapter Y is subject to authorization according to model Z". SFRs are usually not quantifiable: the system either has the functionality or it doesn't.

2. *Quality Attribute Requirements* (QARs) are quantifiable requirements about system quality attributes. They can always be expressed as a number and a scale, e.g. following Gilb's notation techniques [8]. Examples of QARs are reliability, usability, performance and supportability. There are many taxonomies available, e.g. ISO 9126[1].

3. *Implementation Requirements* constitute the third category of supplementary requirements. They put constraints on the system that cannot be measured by system assessment, and incorporate e.g. managerial issues. Examples of IRs are time-to-market, maximum cost, resource availability and outsourcibility. Implementation requirements can be expressed in "-ilities" that make them resemble quality attribute requirements, such as affordability or viability, but they are not about system quality. However, they can be just as important to system design as functional or quality requirements.

A system's compliance with QARs and SFRs can in principle be measured by anyone having access to the system once it has been realized, *regardless of whether they know about its history or its cost*. Compliance with Implementa-

1 Note that a requirement in the ISO 9126 category "Functionality" is not a quality attribute requirement in the NFD model; it can be either a primary or a secondary functional requirement

tion Requirements can only be assessed by looking at how the system was realized.

There is a relationship between Secondary Functional Requirements (SFRs) and functional solutions to Quality Attribute Requirements, which will be discussed later. SFRs can usually be traced back to a high-level quality need, but to express them as a quality requirement would leave too much room for interpretation. For example, the requirement to log system errors over an SMTP interface is an implementation of a manageability need, but to just require that "System management should require at most 0.1 FTE" would allow other, perhaps less desirable solutions. Satisfying Quality Attribute Requirements may also entail adding functionality to the system, but this time the choice of functionality is at the architect's decision.

3.3. The nature of requirement conflicts

The reason for a classification into primary and secondary requirements is a preparation for the NFD process that leads to system decomposition exploiting the requirement conflicts. The NFD version of the leading NFR principle cited above is that *in designing system architectures, the supplementary requirements are more important than the primary requirements.* Primary requirements are never conflicting: if they would, the requirements would be intrinsically inconsistent or the problem statement poorly posed. However, supplementary requirements including secondary functional requirements, can appear to be conflicting, as is explained in the following paragraph.

Requirements on a software system are not intrinsically conflicting, because conflicts arise from limitations in the solutions domain. Boehm and In [4] have based their software tools for identifying quality-requirement conflicts on this fact. We have further analyzed the common solutions to satisfy supplementary requirements, including quality attributes. Our analysis clarifies that some quality attributes and implementation requirements may be so tightly bound to certain types of solutions, that they are effectively inherently conflicting. This situation arises when a quality attribute can only be achieved by one class of strategies, and when this class of strategies is invariably detrimental to another quality attribute. The Feature-Solution graphs introduced in [7] provide a good way to visualize these conflicts. We will illustrate this point with a few examples.

We have categorized the strategies for fulfilling software quality requirements into three types and nicknamed them the three dimensions of software construction: the *process* dimension, the *structure* dimension and the *functional* dimension.

1. One way to achieve supplementary requirements is by making choices in the software building *process*. Models like the Capability Maturity Model and other Software Process Improvement practices generally aim at improving the reliability of software. Recommended practices have been documented to achieve certain quantified Safety Integrity Levels [11] or to fulfill certain security requirements [6]. These practices tend to make the software construction process more expensive, giving rise to the first example of inherently conflicting requirements: *reliability* versus *affordability* (not an NFD quality attribute, but possibly an implementation requirement).

2. Another way to influence quality attributes or to satisfy other supplementary requirements is by making choices in the *structure* of the software. Examples of software structuring solutions include layering, applying of design patterns, choosing higher or lower level languages, modifying the granularity or modularity of the software, and so on. We have started to explore this area somewhat in [13]. Generally speaking, the structure-based solutions seem to have one common element: more structure (i.e. higher level programming language, more layers, higher granularity etc.) means better modifiability, but less efficient code. This is the second example of inherently conflicting quality attributes: *modifiability* versus *efficiency*.

3. The third way to achieve supplementary requirements is by building *functionality* that is specifically aimed at achieving a quality or implementation objective. Examples are encryption and access control functionality to achieve a certain security goal [6], or caching functionality to achieve a certain response time and thus increase usability. Although these types of strategies are not specifically detrimental to other quality attributes, they do increase the size and complexity of the system, leading to effects such as lower affordability and reliability. The reader should note the difference between secondary functional requirements with underlying quality needs and functional solutions aimed at satisfying quality attribute requirements. In the former case, the functional solution is raised to the status of requirement and the responsibility of the requirement specifier. In the latter case, the functional solution is the responsibility of the architect. In practical situations, this dichotomy may be ambiguous and quality needs are translated into functional solutions by an iterative process that involves both the requirements specifier and the architect.

In the above examples of "inherently conflicting" requirements, the conflicts emerge when applying the solution strategies to a single subsystem or component. These conflicts can often be resolved by separating the subsystem or component into different parts, and applying different solution strategies to the respective parts. Viewed from this per-

spective, modifiability and efficiency need not be conflicting: one can decompose a system into low-coupled subsystems for modifiability, and then apply strategies for making the code of each subsystem more efficient. Approaches of this kind are put into practice intuitively by experienced architects, and we have modelled them in our Non-functional Decomposition Framework.

3.4. Applying solution strategies

The NFD model of the architecture process refines the n-dimensional optimization problem of the consensus model into a 3×3 matrix. The cells of this matrix contain strategies from each of the three solution dimensions fulfilling each of the three types of requirements. Whereas the diagonal of the matrix contains obvious strategies (e.g. functional solutions to functional requirements), the off-diagonal cells often suggest important solutions that can help achieve requirements that would otherwise pose problems. Without being complete, we provide some examples of each of the matrix cells.

Functional solutions aimed at functional requirements: the required functions should be implemented.

Functional solutions aimed at quality-attribute requirements: these are functions like encryption, access control, caching and duplication, that are specifically designed to achieve a quality objective.

Functional solutions aimed at implementation requirements: implementation requirements, like outsourcing or time-to-market limitations, often exist. Their realization points to e.g. reuse of off-the-shelf components or purchasing and integrating commercial products.

Structural solutions aimed at functional requirements: examples of structures that contribute to functional requirements are database normalization, extracting of generic functionality, functional or non-functional decomposition.

Structural solutions aimed at quality-attribute requirements: lead to programming in patterns, but there are also other structural strategies contributing to quality attributes, such as the choice of programming language or correct parametrization. The NFD method itself also contributes to fulfilling quality-attribute requirements.

Structural solutions aimed at implementation requirements: implementation requirements like preferred release schedules can be realized by adapting the structure of the system to accommodate incremental deployment. Another example is the choice of a rapid development platform (fourth generation language), which dictates a particular system structure. NFD can also be applied here.

Process solutions aimed at functional requirements: an example is using a conventional cascade-development method, which prioritizes system functionality over time and budget limitations.

Process solutions aimed at quality-attribute requirements: examples of these are best practices from the Software Process Improvement community to improve reliability, or Common Criteria assurance packages to achieve security goals.

Process solutions aimed at implementation requirements: implementation requirements such as "user involvement" can be realized by prototyping, or "strict deployment deadline" by using a development method such as EVO or RAD that employs time-boxing techniques.

The combination of applied solutions in the process dimension results in the best *development process* to fit the system requirements. The sum of the applied functional solutions and the realization of the primary functional requirements together compose the *system functionality*. The *system architecture* consists of a high-level description of the applied functional (logical view) and structural (subsystem, development, deployment views) solutions.

3.5. The role of the NFD process

Non-Functional Decomposition (NFD) is a strategy in the structural dimension of software construction. The NFD process helps to optimize the structure of the system for all supplementary requirements, including implementation and secondary functional requirements, which are generally associated with process or functional solutions first. It does this by adapting the system structure to the requirement conflicts in the system, and isolating conflicting requirements in subsystems that can then be individually optimized by applying process, structural or functional strategies. It is essentially an iterative divide-and-conquer strategy for resolving requirement conflicts.

4. The NFD process

The process of NFD is depicted in Figure 3 and contains the following steps.

Gather and prioritize requirements can be based on any modern requirements elicitation technique, provided that the documented requirements show how the Primary Functional Requirements (PFRs) are mapped to the Supplementary Requirements (SRs). It is important that the requirements are somehow prioritized, since prioritization of PFRs is important for project and release planning, and prioritization of SRs is important for the architecture. *Example from the Unified Process:* in the UP, FRs are generally documented as use cases, and SRs as supplementary specifications. The NFD method requires that the supplementary specifications are made specific to (groups of) use cases, e.g. by documenting them in the Use-Case Descriptions, or specifying to which use cases SRs apply in the Supple-

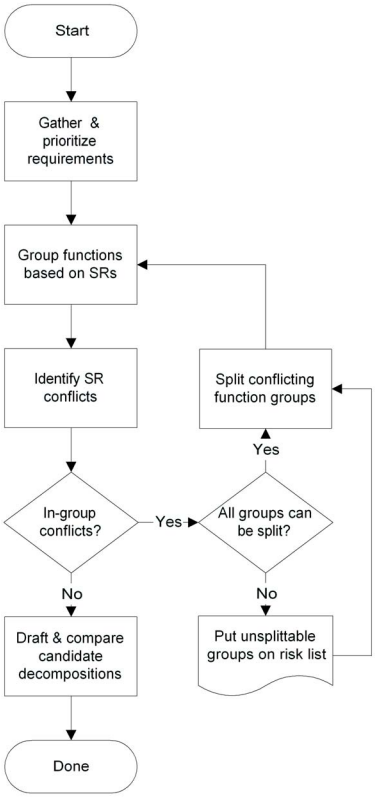

Figure 3. The NFD Process.

mentary Specifications document. Another way of linking SRs to FRs is the use of so-called Specific Scenarios as described in [2].

Group functions based on supplementary requirements is the process of finding all (primary) functional requirements that share or have similar supplementary requirements, and grouping them together. This will yield a number of cross-sections of the functionality of the system, depending on which supplementary requirement is used as a grouping criterion. In this step, the distinction between PFRs and SRs is less important: each group will have a number of functions, originating from both primary and secondary requirements, which will be treated equally during the remainder of the process. *Example:* a time-to-market priority grouping will divide functionality into groups that are candidates to be included in different release phases of the system, while availability grouping will divide functionality into candidate groups to run on platforms with differing availability characteristics.

Identify supplementary requirement conflicts yields two types of conflicts:

1. *Grouping conflicts* are caused by differences in grouping of functions, i.e. the grouping of the functions is significantly different from one SR to the other. *Example:* there are three function groups, called WorkFlow, DataEntry and Analysis (Figure 4). Security requirements for DataEntry and Analysis are similar and more restrictive then those for WorkFlow, but modifiability requirements for Analysis are more stringent than those for DataEntry and WorkFlow.
2. *In-group conflicts* are conflicting supplementary requirements within one function group. *Example:* the Analysis function group from the previous example has both critical performance requirements and high modifiability requirements.

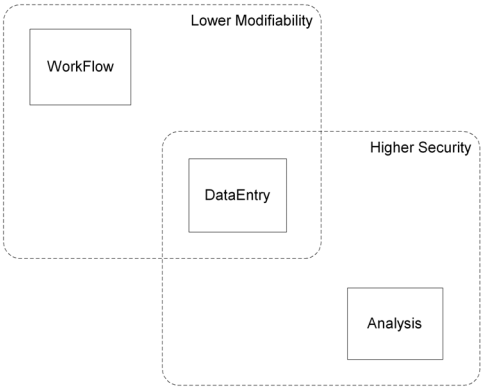

Figure 4. Grouping conflict example.

Split conflicting function groups deals with in-group conflicts. Most of the time, a further analysis of an in-group conflict will show that the conflicting requirements can actually be assigned to different functions. These functions are then separated, leading to a splitting of the function group. The resulting two or more new function groups may then be reconsidered for being included in other function groups, so the process re-enters the "Group functions based on SRs" stage. This loop is repeated until no in-group conflicts are left that can be split further. Function groups that cannot be split in any way are flagged as risk factors. They deserve close attention during the rest of the process and need to be dealt with prior to large-scale project implementation, e.g. in an architectural prototype.

Draft and compare candidate decompositions: After the in-group conflicts are solved, the resulting grouping conflicts will be the basis for the architectural decomposition. A number of candidate decompositions into architectural components result, each favoring the main supplementary requirement that the function grouping is based upon.

At this stage, prioritization of supplementary requirements becomes important. In our experience, the candidate decomposition that is based on the SRs having the highest stakeholder priority, yields the architecture that best fits the stakeholder requirements. This does not mean that we suggest that the n-dimensional optimization problem men-

tioned earlier can be reduced to a series of one-dimensional optimizations, designing for the most important requirement first and then narrowing down the design choices further for each requirement. But in the decomposition process, it turns out that the decompositions based on the SRs with the highest priority have the best chance of yielding a system the stakeholders can live with. In case of doubt, an impact analysis of the top three decompositions can be made, e.g. by using the CBAM method [1]. The decomposition with the best fit to the stakeholder's needs considering the total cost of ownership is selected for implementation.

5. Case: criminal investigation system

5.1. Background

Two IT organizations affiliated with the Dutch ministry of internal affairs, the ISC and the CIP, are currently developing a product line for nationwide processing of and access to criminal investigation and intelligence data.[2] The product line is called Politiesuite Opsporing (PSO, "Police Suite for Investigation"). One of the authors is the lead software architect for PSO, and NFD was used to design the suite's top-level decomposition. The main challenge was to create an architecture that would allow the addition of many new products to the suite in the years to come, without compromising the strict privacy and confidentiality requirements on the system.

5.2. Summary of requirements

The suite's primary functionality is the support of all business processes related to criminal investigation, including management of the processes, gathering of data through multiple channels, and structuring and analysis of the data. The three most important supplementary requirements according to the stakeholders are:

SR1 *Authorization:* access to criminal investigation data is restricted by special privacy laws. Unauthorized access to privileged data is by far the biggest threat to a criminal investigation system.

SR2 *Reliability:* reliable application of authorization and other business rules is crucial. The system should be designed in such a way that the enforcement of especially authorization rules is reliable and stable, even after several product generations.

SR3 *Development time:* the criminal investigation systems currently in use are based on obsolete architectures and there is an urgent need in the field to support new functionality. Exceeding the stated deadline of one year of development time is unacceptable.

SR1 is a secondary functional requirement, SR2 a quality attribute requirement, and SR3 an implementation requirement.

5.3. Results

NFD was applied by first mapping the most important supplementary requirements onto the functional features, and then basing the main architectural decomposition on this mapping. SR1 applies specifically to the data gathered for criminal investigation purposes. It turned out that the most reliable and best maintainable solution for the future was to create a central component for access control and storage of these data. Since the legal name for storage of such data is a police register, this central component was named the "register vault". By using off-the-shelf components supplied by a database vendor, the register vault could be assembled and an architectural prototype evaluated within a few months time, making a good start at satisfying SR3.

In this case, the Rational Unified Process®[3] (RUP®) was used to streamline the development process. The RUP Supplementary Specifications artifact is the place to document quality and other supplementary specifications, but the standard template treats these as system wide or "general" requirements. We changed the template slightly to accommodate documenting the mapping between primary and supplementary requirements. We did this at the level of "features" as defined by RUP. This allowed us to document the trace from primary and supplementary requirements to the system decomposition design decisions.

In the end, the Non-Functional Decomposition principles turned out to be very useful in communicating to the stakeholders how our design decisions were related to their stated supplementary requirements.

6. Case: Dutch road-pricing system

In this section we will apply the NFD model and process to analyze a large system on roadpricing. One of the authors was involved in this project (until it was suspended for political reasons), which is described below. Although NFD was not available at the time, applying the method retrospectively to this case presents a good illustration of its principles.

2 The authors would like to thank the ISC and CIP organizations for granting permission to publish this case.

3 RUP, Rational and Rational Unified Process are trademarks of International Business Machines Corporation.

6.1. Background

In the late 1990s, the Dutch government decided to drastically change the tax system for automobile owners and drivers. Automobile tax traditionally consisted of gasoline tax, annual motor vehicle tax and BPM (personal motor vehicle tax payed once when purchasing a vehicle). The system should be replaced by a direct tax system based on usage: gasoline tax would be reduced to the legal European minimum, and annual motor vehicle tax and BPM would be completely replaced by a roadpricing scheme called "Kilometerheffing" ("Charging by kilometer"), hereafter referred to as KMH. By differentiated pricing of road segments based on the time of day and location, the scheme could also be used to make drivers avoid congested (and thus more expensive) areas during rush hours. Feasibility of the scheme would highly depend on the use of IT systems, some of which would have to be in the vehicle. The government planned to share the cost of developing and manufacturing the in-vehicle systems (called "Mobimeters") with the multimedia, communication and automobile industries by making generic components of those systems available to those industries.

6.2. System requirements

The Mobimeter requirements were published[4] in order to facilitate discussion with industry partners. Without going into too much detail, the original requirements can roughly be summarized as follows.

PF1 The system shall continuously measure the position and driving direction of the vehicle.

PF2 The tariff used for calculating the cost during a journey is determined on the basis of the following parameters (hereinafter referred to as Tariff Parameters):

- Date and time;
- Vehicle position;
- Direction of travel;
- Tariff table;
- Vehicle category.

PF3 A driver shall be notified of the applied tariff while driving.

PF4 The system shall determine the distance travelled by a vehicle.

PF5 The mobility costs due is calculated as being the product of the distance travelled times the current tariff.

PF6 At least once every month in which 1000 kilometers has been driven or at least once per elapsed year, whichever comes earlier, all data shall be communicated to the tax office.

PF7 The system shall be able to receive new tariff tables.

The associated supplementary requirements (SRs) are summarized below. For brevity, we only mention the most important ones:

S1 *Privacy:* a vehicle's mobility patterns may not be deducible, either in real time (tracking) or afterwards from system data (tracing).

S2 *Verifiability:* the KMH Road-Pricing System shall enable verification that the road pricing charge has been determined correctly, without requiring more than one physical inspection per year.

S3 *Provability:* The system shall enable drivers to verify the correctness of the charges by inspecting all relevant data.

S4 *Security:* All data required for the KMH Road-Pricing System process will be protected against unauthorized modification.

S8 *Re-usability:* all in-vehicle system functions that could be useful for other applications shall be made available for re-use by third parties.

S12 *Viability:* industry partners and the relevant governmental and non-governmental organizations shall be involved as much as possible in the development of the KMH system.

S13 *Standardization:* any interfaces to be designed for in-vehicle equipment shall be developed in close cooperation with the relevant standardization bodies.

S1, S4 and S8 are quality-attribute requirements. S2 and S3 are secondary functional requirements. S12 and S13 are implementation requirements that originally did not occur in the requirements document, but in the project plan. We mention them here because in the NFD model they qualify as supplementary requirements. As will be shown, they did impact the system architecture.

Table 1 shows how the original supplementary requirements map onto the primary requirements. Note that some of the mappings apply to subsets of a particular primary functions only, e.g. standardization of *only* the interface, or protection of *only* the data:[5] a finer granularity of functionality will help us split and re-group the functions later on. The next step in the mapping process would be to split up

4 The original requirements were drawn up by a team led by Maarten Boasson (University of Amsterdam). They were based on the "Mobimiles" report of Roel Pieper (University of Twente), which was never formally published

5 The presence of supplementary requirements that apply to data elements or storage is quite common in our experience; these requirements often lead to special data storage components, especially if they have high priority.

	PF1	PF2	PF3	PF4	PF5	PF6	PF7
S1	X					X	
S2	X	X		X	X	X	
S3	X	X		X	X	X	
S4	X	X		X	X	X	X
S8	X		X			X	X
S12	X		X			X	X
S13						X	X

Table 1. Original mapping of supplementary onto primary requirements.

	PF1	PF2	PF3	PF4	PF5	F6a	F6b	F6c	PF7
S1	X						X	X	
S2	X	X		X	X	X	X		
S3	X	X		X	X	X		X	
S4	X	X		X	X	X	X	X	X
S8	X		X			X	X	X	X
S12	X		X			X	X	X	X
S13						X	X	X	X

Table 2. Second iteration mapping of supplementary onto primary requirements.

the PFs to achieve a more exact mapping, but that exercise would be too detailed for this paper.

A number of grouping strategies present themselves, but let us first look at the glaring in-group conflict concerning privacy versus verifiability and provability in the group PF1+PF6. According to NFD, we split the group to resolve the conflict. Clearly, if all data on which the charge is based are communicated to the tax office, the privacy requirement is violated. The objective is to split PF6 in such a way that only less privacy-sensitive data are communicated and simultaneously maintain the verifiability. After checking back with the stakeholders for the business-need behind PF6, it turns out that we can split as follows

F6a At least once every month in which 1000 kilometers has been driven or at least once per elapsed year, whichever comes earlier, the total charges and the total distance per tariff shall be communicated to the tax office.

F6b Spot checks: a travelling vehicle shall be able to answer challenges made by roadside-enforcement equipment by transferring all currently measured data and the current tariff table.

F6c On request by the driver, the system shall communicate all data used to calculate charges to him.

F6a is sufficient to fulfill PF6's underlying need; F6b is a functional solution to S2, and F6c is a functional solution to S3. F6b adds a short-range communication function to the Mobimeter, which falls under the re-usability and standardization requirements S8, S12 and S13. This analysis leads to a new mapping table.

Table 2 shows the SR/FR mapping after splitting PF6. The conflict between S1 and S2 is now isolated in component F6b, the spot check function. This reflects the issue that privacy-sensitive data are present in the spot-check equipment. We put this conflict aside as a risk that will be managed by a protocol surrounding the management of these data: how long they may be stored, to what purpose, etc.

Let us now look at grouping criteria. According to the stakeholders, security, privacy and viability through

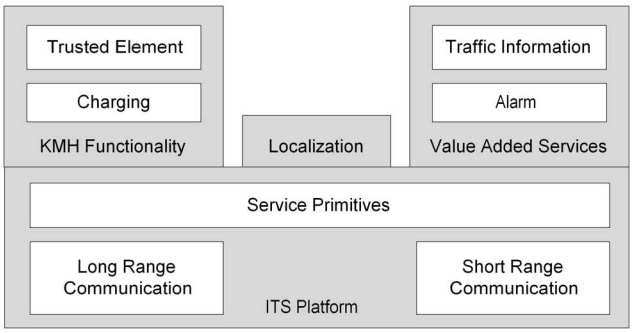

Figure 5. Mobimeter architecture.

re-usability are the most important supplementary requirements and in that order of priority. The security requirement S4 groups the associated data of all functions except the driver display, which basically means that the Mobimeter should contain a secure data storage component or "trusted element" that all KMH functions should have access to. Privacy requirement S1 no longer applies to F6a, since we split off the data from which the mobility pattern can be deduced. So S1 now groups PF1, F6b and F6c. S1/F6b becomes the basis for storage requirements on the roadside spot-check system, and the PF1/F6c group leads to a subsystem called the "user log".

Finally, the viability and re-usability requirements S8 and S12 group the communications functions of F6a/b/c and PF7, the display of PF3 and the vehicle localization of PF1 into a subsystem called the "In-vehicle Telematics Services platform". This platform is designed to have standardized interfaces (S13) both to the KMH-specific part of the in-vehicle equipment and to any additional (optional) service-related components such as a navigation system. It gives access to services like GPS localization and long-range and short-range digital communication. The final system decomposition is shown in Figure 5. In the final version, the localization function was separated from the ITS platform in order to fulfill an additional reusability requirement on GPS equipment, which was already supplied in some cars.

7. Conclusions and discussion

In this paper, we have presented the Non-Functional Decomposition (NFD) model as a technique to bring more clarity and structure in the mapping of requirements onto a system architecture. The key of our technique is to split the requirements into primary and supplementary requirements, and to create a mapping between those categories. The NFD process helps in optimizing the structure of the system for all supplementary requirements, including implementation and secondary functional requirements. NFD adapts the system structure to the requirement conflicts in the system and isolates conflicting requirements in subsystems that can then be individually optimized by applying process, structural or functional solution strategies of which examples were presented.

The PSO product line was presented as a case study of the application of NFD. The main result here was a well documented traceability between supplementary requirements and system decomposition design decisions. This traceability supported the project team in communicating to the stakeholders the effects of their stated requirements, and the rationale behind the main design decisions.

The KMH project was used as another case study, where we have indicated how the Mobimeter architecture as it was published, can be reconstructed with the NFD model. The details of the iterations and the break-down of all functional requirements needed to make the exact mappings cannot be printed here because of space limitations. The examples of resolving in-group conflicts and grouping functions according to supplementary requirements were given to show the application of the method and principles of NFD.

The validity of the observations on architecting is not only confirmed by our daily work, but can be easily verified by evaluating successful architectures like client/server or n-tier architectures. The components in these architectures all differ in their supplementary behavior, and display specific geographical accessability, modifiability, efficiency or portability attributes.

We view NFD as a framework that uses the observations made in section 2 to improve the architecting process. These observations are not new, we believe they have always been implicit in the work of experienced designers. By making them explicit, NFD makes the architecting process of transforming requirements into system design more reproducible, more transparent and more reliable. Future work is in further application of NFD in actual technically complex projects, and in the exploration of other areas in which it could be deployed.

References

[1] J. Asundi, R. Kazman, and M. Klein. Using economic considerations to choose among architecture design alternatives. Technical Report CMU/SEI-2001-TR-035, SEI, 2001.

[2] F. Bachmann, L. Bass, and M. Klein. Deriving architectural tactics: A step toward methodical architectural design. Technical Report CMU/SEI-2003-TR-004, SEI, 2003.

[3] F. Bachmann, L. Bass, and M. Klein. Preliminary design of arche: A software architecture design assistant. Technical Report CMU/SEI-2003-TR-021, SEI, 2003.

[4] B. Boehm and H. In. Identifying quality-requirement conflicts. *IEEE Software*, pages 25–35, March 1996.

[5] J. Bosch. *Design and Use of Software Architectures*. Addison Wesley, 2000.

[6] CCPSO. Common criteria for information technology security evaluation. http://www.commoncriteria.org/cc/cc.html, August 1999. Produced by the Common Criteria Project Sponsoring Organizations, version 2.1.

[7] H. de Bruin and H. van Vliet. Top-down composition of software architectures. In *Proceedings of the 9th Annual IEEE International Conference on the Engineering of Computer-Based Systems (ECBS)*, April 2002.

[8] T. Gilb. *Principles of Software Engineering Management*. Addison Wesley, 1988.

[9] P. Gruenbacher, A. Egyed, and N. Medvidovic. Reconciling software requirements and architecture: the cbsp approach. In *5th IEEE International Symposium on Requirements Engineering*, IEEE CS, August 2001.

[10] P. Gruenbacher, A. Egyed, and N. Medvidovic. *Journal for Software and Systems Modeling*, to appear.

[11] IEC. 61508: Functional safety of electrical/electronic/ programmable electronic safety-related systems, 1999.

[12] B. Paech, A. Detroit, D. Kerkow, and A. von Knethen. Functional requirements, non-functional requirements, and architecture should not be separated - a position paper. REFSQ, Essen, Germany, September 2002.

[13] E. R. Poort and P. H. N. de With. Modelling the relationship between quality attributes and architecture of software intensive systems. PROGRESS, October 2003.

0-7695-2172-X/04 $ 20.00 © 2004 IEEE