Nonfunctional Requirements: From Elicitation to Conceptual Models

Luiz Marcio Cysneiros, *Member*, *IEEE Computer Society*, and
Julio Cesar Sampaio do Prado Leite, *Member*, *IEEE Computer Society*

Abstract—Nonfunctional Requirements (NFRs) have been frequently neglected or forgotten in software design. They have been presented as a second or even third class type of requirement, frequently hidden inside notes. We tackle this problem by treating NFRs as first class requirements. We present a process to elicit NFRs, analyze their interdependencies, and trace them to functional conceptual models. We focus our attention on conceptual models expressed using UML (Unified Modeling Language). Extensions to UML are proposed to allow NFRs to be expressed. We will show how to integrate NFRs into the Class, Sequence, and Collaboration Diagrams. We will also show how Use Cases and Scenarios can be adapted to deal with NFRs. This work was used in three case studies and their results suggest that by using our proposal we can improve the quality of the resulting conceptual models.

Index Terms—Software design, requirements elicitation, nonfunctional requirements, goal graphs, UML conceptual models.

1 Introduction

Software systems, aside from implementing all the desired functionality, must also cope with nonfunctional aspects such as: reliability, security, accuracy, safety, performance, look and feel requirements, as well as organizational, cultural, and political requirements. These nonfunctional aspects must be treated as nonfunctional requirements (NFRs) of the software. They should be dealt with from the beginning and throughout the software development process [9], [10].

Ineffectively dealing with NFRs has led to a series of failures in software development [5], [26], including the very well-known case of the London Ambulance System [17], where the deactivation of the system right after its deployment was strongly influenced by NFRs noncompliance. Literature [7], [15], [11] has been pointing out these requirements as the most expensive and difficult to deal with.

In spite of their importance, NFRs have, surprisingly, received little attention in the literature and are poorly understood compared to less critical aspects of the software development [10]. The majority of the work on NFRs uses a product-oriented approach, which is concerned with measuring how much a software system is in accordance with the set of NFRs that it should satisfy [25], [2], [16], [31], [30].

There are, however, a few that propose to use a process-oriented approach in order to explicitly deal with NFRs [10], [25], [3], [41]. Most of these works propose the use of techniques to justify design decisions on the inclusion or exclusion of requirements that will impact the software design.

Unlike the product-oriented approach, our approach is concerned with making NFRs a relevant and important part of the software development process. It is also possible to find standards [22], [39], [33] that can offer some guidance on eliciting NFRs. However, these standards basically give different taxonomies for some of the NFRs. The elicitation process per se is shallow. There is also a lack of guidance on how one might integrate the NFRs into design.

We propose a strategy to elicit NFRs and guide the software engineer to obtain conceptual models that will have explicit traces to the NFRs and vice-versa.[1] Our elicitation process is based on the use of a lexicon that will not only be used to anchor both functional and nonfunctional models, but also to drive NFR elicitation. A lexicon representing the common vocabulary used in the domain is built. Later, NFRs are added to this lexicon. Possible solutions for implementing these NFRs are also added to the lexicon.

The lexicon will then drive the construction of NFRs graphs [8] slightly extended to fit our strategy. NFR graphs are and/or graphs that decompose nonfunctional requirements, from vague abstractions to more concrete descriptions. Heuristics for conflict detection are then used to guide the NFRs reasoning. Finally, the strategy provides a systematic way of integrating the elicited NFRs into use cases and scenarios as well as class, sequence and collaboration diagrams. The integration process can also be used to validate ongoing projects in such a way that, even if one has a project where the conceptual models are

- L.M. Cysneiros is with the Department of Mathematics and Statistics, Information Technology Program, York University, 4700 Keele St., M3J1P3, Toronto, Canada. E-mail: cysneiro@mathstat.yorku.ca.
- J.C.S.d.P. Leite is with the Departamento de Informática, Pontificia Univiversidade Católica do Rio de Janeiro, R. Marques de São Vicente, 225 Rio de Janeiro, Brasil 22453-900. E-mail: julio@inf.puc-rio.br.

Manuscript received 12 Aug. 2002; revised 30 Aug. 2003; accepted 1 Mar. 2004.
Recommended for acceptance by J. Knight.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 117119.

1. Our work is based on the results of the doctoral dissertation of Dr. Cysneiros and on our ongoing research on the aspects of nonfunctional requirements. This journal paper builds upon, and uses the results of, other published materials [11], [12], [14], but provides a new and integrate view of our results.

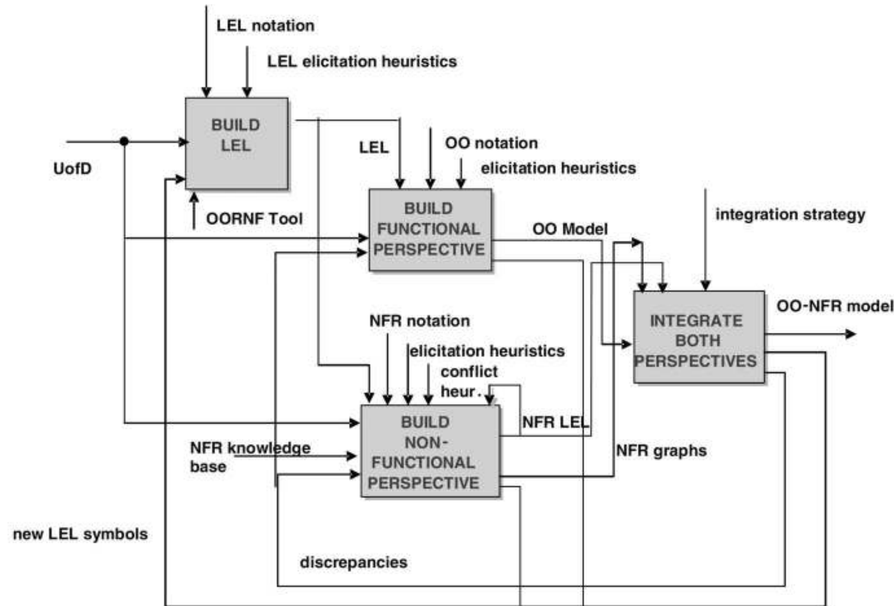


Fig. 1. An overview of the strategy to deal with NFR.

already designed, he can still integrate them with NFRs. In the same way, the integration process can also be used for enhancing legacy systems.

We carried out three case studies to test the proposed strategy. Two of these case studies were controlled experiments that were based on the use of the specification for the implementation of the Light Control System for the University of Kaiserslautern [4]. This system has to control all the lights in a building using several sensors so the system can detect the presence or the absence of people in a room, as well as the amount of exterior illumination coming through the windows. This way, the system can maintain the illumination of a room in accordance with several criteria defined in the system specification. The third case study was conducted during the development of a software system for controlling a clinical analysis laboratory. This case study was carried out within a software house that was developing a real system to control the whole production process of a laboratory, ranging from drawing the blood to delivering the patient's report to the patient. Section 5 will present further details.

The results of the case studies suggest that the use of the proposed strategy can lead to a more complete conceptual model, as well as to a faster time to market process since NFRs related errors can be avoided. Confronting these results with the measured overhead to apply the strategy suggests that the proposed strategy can lead to a more productive process.

Section 2 will present an overview of the entire strategy to deal with NFRs, while Section 3 will tackle NFRs elicitation. Section 4 will detail how to represent NFRs in Use Case Diagrams and Scenario Models and how to integrate NFRs into class, sequence, and collaboration diagrams. Section 5 will present the results from three case studies and Section 6 concludes the paper.

2 An Overview of the Strategy to Deal with NFR

We view the software development process as being composed of two independent evolutionary perspectives. As one perspective focuses on the functional aspects of the software system, the other deals with nonfunctional aspects. The rationale to have two different processes is twofold; first, it allows the process to be used in legacy applications; second, we are using an evolution perspective. Requirements changes are usually motivated by either functional or nonfunctional aspects, so treating them separately eases the evolution aspect. A positive side effect of using two distinct processes is that we can analyze nonfunctional graphs to check for design consistency among the several interdependencies among NFRs. For instance, if we have security demands, like using cryptography, it will have to be consistent with our demands for spacing and performance.

As depicted in Fig. 1, we propose to build both the functional and the nonfunctional perspectives anchored in the Language Extended Lexicon (LEL) [26]. This policy assures that a common and controlled vocabulary will be used in both functional and nonfunctional representations. Note that, in Fig. 1,[2] LEL is a control arrow to both BUILD FUNCTIONAL PERSPECTIVE and BUILD NONFUNCIONAL PERSPECTIVE as such, traceability is enabled by construction since both perspectives will be anchored on the same lexicon.

From Fig. 1, we can see that there are four major activities in our strategy. We first build the lexicon, and then build the functional and the nonfunctional models concurrently. We integrate the two models and attend to the different feedbacks during the evolution of the process (see the arrows "new LEL symbols" and "discrepancies").

2. In a SADT model [36], the control arrow determines how the activity will be performed.

The First step is to build the lexicon, which is a natural language-oriented front-end to the strategy, and the anchor for the vocabulary used in the software. The lexicon representation is based on the Language Extended Lexicon (LEL) [26]. LEL registers the vocabulary of a given Universe of Discourse (UofD[3]). It is based upon the following simple idea: understand the problem's language without worrying about deeply understanding the problem. As such, we assure that the vocabulary is well anchored. LEL entries have to be defined both in terms of their denotation as well as the particular connotation in that given context. It is also structured in a way that their entries are naturally linked in a hypergraph (see Fig. 3). LEL is first produced without focusing on nonfunctional aspects. These aspects are added to LEL at the BUILD NONFUNCTIONAL PERSPECTIVE. Note in Fig. 1 this activity has as input "NFR knowledge base" and the control/output "LEL with NFR."

Once having the first LEL, without the NFR related information, we may start building the functional model (BUILD FUNCTIONAL PERSPECTIVE). We do not detail this process in the paper, which can be any OO modeling formation method,[4] but with the caveat explained before, all the elements of the model must follow the same vocabulary as used in the lexicon.

In parallel with BUILDING FUNCTIONAL PERSPECTIVE, we will also BUILD NONFUNCTIONAL PERSPECTIVE. In this activity, we will add the desired NFRs to the existing or recently created LEL. For this purpose, LEL may also express that one or more NFR is needed by an entry. It is structured to handle dependency links between one NFR and the entries related to it.

After including NFRs in the lexicon, which shows all the desired NFRs and some of their operationalizations, we represent these NFRs in a set of NFR graphs (BUILD NONFUNCTIONAL PERSPECTIVE) using Chung's NFR framework[5] [8], [10], [32]. The framework proposes to use nonfunctional requirements to drive design and to support architectural design. NFRs are viewed as goals (roots of an and/or graph) that are decomposed into subgoals (subgraphs) until all the necessary actions and information are represented at the leaf levels of the graphs. These actions and information are called operationalizations.

The NFR framework allows a deeper level of refinement and reasoning about NFRs. Once the graphs have been built, we then apply a series of heuristics to find interdependencies between graphs and try to solve any possible conflicts. As such, there is a feedback from this activity to the first activity of our SADT model (Fig. 1). However, this framework does not detail an elicitation process or NFR integration into design. Two of our previous works [11], [14] reinforce the premise presented by Chung [10], that the lack of integration of NFRs to functional requirements may lead to incomplete conceptual models. During acceptance or after deployment these NFRs will likely be demanded by customers, and the lack of integration may result in projects that will take more time to be concluded as well as in bigger maintenance costs.

Finally, we need to integrate the functional and nonfunctional perspectives. The integration process can take place either in the early phases, integrating the NFRs into the use case or scenario models, or later, integrating NFRs into class, sequence and collaboration diagrams. Actually, the best case scenario happens when the integration takes place in both the early and late phases. Doing so, one could not only check if NFRs are correctly represented in later phase models, but also reevaluate these NFRs under the viewpoint of new design decisions or requirements evolution that may have happened. Again, it is important to stress that the integration is based on the idea that the NFR graphs and class, sequence and collaboration diagrams will be built using LEL symbols.[6] For each class, we search all of the NFR graphs looking for occurrences of the symbol that is named after that class. For each match, we must see if the operationalizations for this NFR are already implemented in the class. If they are not, we should add the necessary operations and/or attributes. For use cases and scenarios, a similar approach will be used. If a class is not named after a LEL symbol, either a synonym is missing in LEL or LEL is incomplete and should, therefore, be updated. Once LEL is updated, all of the steps in the nonfunctional perspective should be carried out again, as well as a new integration into the design models. This vocabulary scheme produces a natural traceability that helps to navigate among models, enhancing the ability to check possible impacts that could arise from changes in the design. These changes may also come from a scenario where requirements had evolved and hence demanded new trade off analysis.

It is important to emphasize that we do not propose to deal with NFRs elicitation within the scope of the functional perspective. In order to attain NFRs, their goal graph usually requires a very detailed reasoning to reach their operationalizations (graph leafs). In general, we start from a very abstract notion of NFR, e.g., Performance, and start refining it into more specific ways of achieving this goal, for example, Space Performance and Time Performance. This process continues until we reach a level where the appropriated action or data will be sufficient to operationalize this goal, e.g., Use Cryptography if we consider the Space Performance.

Another reason for dealing with NFRs separately, is that NFRs frequently present many interdependencies with other NFRs that might require trade offs among different possible design decisions. Take for example the Use of Cryptography mentioned above. While it may be used to achieve Space Performance goals, it may conflict with Time Performance goals. Dealing with these particular aspects of NFRs can be quite difficult and confusing using only use cases, scenarios, or even class diagrams. Note that, in our approach, one does not discard one solution for another. The software engineer must evaluate the contributions of each alternative and decide which one would be taken, but

3. "The overall context in which the software will be developed and operated. The UofD includes all the sources of information and all the people related to the software. It is the reality reviewed by the set of objectives established."

4. E.g., the RUP Process [23] (6).

5. As pointed out by one of the referees, the NFR Framework can be understood as a five-tuple <G,L,GM,R,P>, where G is a set of rules, L is a set of link type, GM is a set of generic methods, R is a collection of correlation rules, and P is a labeling procedure.

6. This is the caveat mentioned above, when describing BUILD FUNCTIONAL PERSPECTIVE.

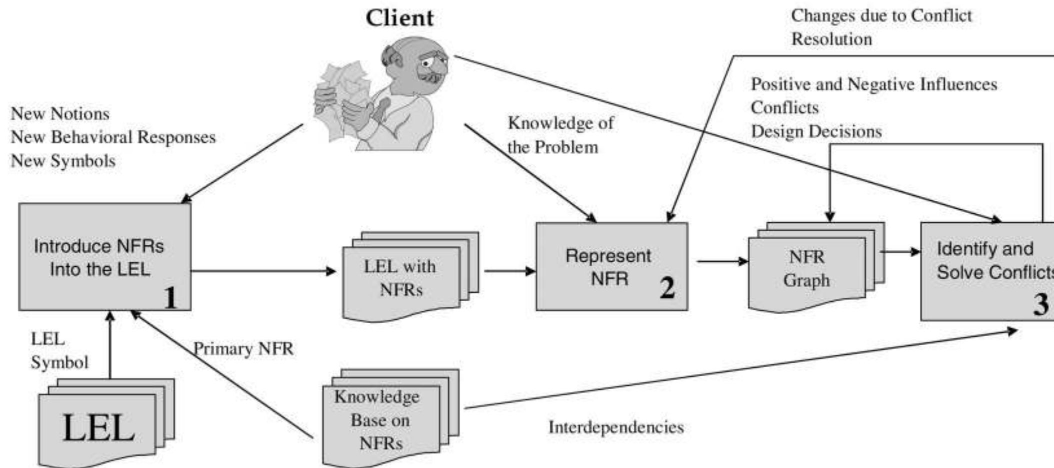


Fig. 2. Building the nonfunctional perspective.

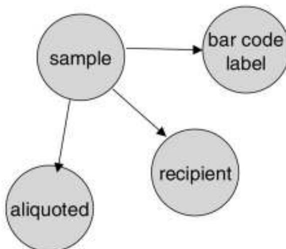
the other alternatives will still be modeled and hence the rationale for decisions will be kept.

3 Building the Nonfunctional Perspective

As mentioned in Section 2, for building the nonfunctional perspective (Fig. 2), we use an existing LEL or, in a case where it does not yet exist, we build a new one. We must add to the existing, or recently created LEL, the NFRs that are desired by customers. Once we have the lexicon showing all of the desired NFRs and some of their operationalizations, we represent these NFRs in a set of NFR graphs using the NFR Framework [8] extended with a few new features. The NFR framework allows us to model and reason about NFRs at a deeper level of refinement than within LEL. Heuristics are then applied to search for possible interdependencies, either positive or negative. Possible conflicts must be negotiated with stakeholders to reach a compromise that represents an agreement on a possible solution that addresses all concerns.

Further details will be showed in the next sections.

(a)



(b)

Fig. 3. (a) Example of a LEL entry. (b) LEL as a graph.

3.1 Using LEL to Support NFRs Elicitation on Earlier Phases

The decision of using LEL was threefold. First, it is a natural language representation, and layman has no difficulty in dealing with it, at least this is our experience with using LEL for the last 10 years. Second, it is structured as a hypertext graph (Fig. 3b) in which every node, the lexicon entry, has both denotations, and connotations adding contextual semantics to the vocabulary. It is used as an anchor for further software representations, thus providing a natural traceability feature among distinct models.

LEL is based on a code system composed of symbols, where each symbol is an entry expressed in terms of notions and behavioral responses. The notions must try to elicit the meaning of the symbol and its fundamental relations with other entries. The behavioral response must specify the connotation of the symbol in the UofD, i.e., what happens because this symbol exists. Each symbol may also be represented by one or more aliases.

LEL construction must be oriented by the minimum vocabulary and the circularity principles. The circularity principle prescribes the maximization of the usage of LEL symbols when describing LEL entries, while the minimal vocabulary principle prescribes the minimization of the usage of symbols exterior to LEL when describing these symbols. Because of the circularity principle, LEL has a hypertext form. Fig. 3a shows an example of an entry in LEL.

The underlined words/expressions are other LEL symbols. Fig. 3b shows the hypertext as a graph, with each node being a lexicon entry.

Since LEL is not a function-oriented description, it has entries that refer both to the functional and to the nonfunctional perspectives.

Although LEL can contain nonfunctional aspects of the domain, at least the very first version of LEL is usually mainly composed of symbols related to functional requirements. This is due to the very vague nature of nonfunctional requirements and, quality aspects, in spite of their importance, are usually hidden in everyone's minds. However, it does not mean that the software engineer cannot register information about nonfunctional requirements if the opportunity arises. A well-defined set of symbols representing the vocabulary of the UofD is an important step to be taken.

We have extended LEL to help with NFRs elicitation. LEL is now structured to express that one or more NFR is needed by a symbol. It is also structured to handle dependency links between one NFR and all the notions and behavioral responses that are necessary to consider this NFR. This is stressed in Fig. 2 since the first step to building the nonfunctional perspective is to enhance the existing LEL with the customer's NFRs. To do that, we run through all LEL symbols using an NFRs' knowledge base, to ask ourselves, and the customers if any of the NFRs in this knowledge base may be necessary to each of the LEL symbols.

Each NFR expressed as desirable by customers is represented as a notion for this symbol using the pattern "Has NFR"+NFR. To each NFR expressed, we must again refer to the knowledge base trying to find possible refinements and operationalizations to this NFR. Operationalizations found must then be represented as notions or behavioral responses either in this symbol, or eventually in other symbol. In some cases, operationalizing an NFR may call for adding features to symbols other than the one currently being analyzed, or even to create a new symbol. Once the operationalization is introduced, it is necessary to introduce a traceability link in the notion or behavioral response pointing to the NFR that originated this operationalization. The traceability link will follow the pattern: "NocaoOrg[" + LEL symbol + "&" + NFR + "&" + internal number]. The string "NocaoOrg" is used to differentiate this entry so one can clearly see that this notion or behavioral response exists to operationalize a NFR present in "LEL symbol." This link will help later on when NFR graphs will be produced. It will also help the software engineer whenever dealing with the impacts of satisficing or not satisficing an NFR.

We have built knowledge bases on some NFRs. Up to now we have created knowledge bases in the form of catalogues for: Usability, Traceability, and Privacy. Those can be reached at: http://www.math.yorku.ca/~cysneiro/nfrs.htm. We intend to keep updating these catalogues as well as creating new catalogues as we gain comprehensive practical experience in other NFRs.

Both LEL representation and the extension for dealing with NFRs are implemented in the OORNF Tool [34]. In this tool, we find an NFR knowledge base that can, and must be constantly updated. This knowledge base stores a variety of NFRs, and some common ways to decompose them. The tool brings along with the definition of these NFRs a set of possible conflicting NFRs as well as a set of NFRs that may be positively impacted by this NFR. The OORNF tool supports LEL, scenarios [29], [20], and CRC cards [42] editing. Our use of the tool here is just to help the presentation of the steps and the products we use; it is not meant to be the focus of the paper.

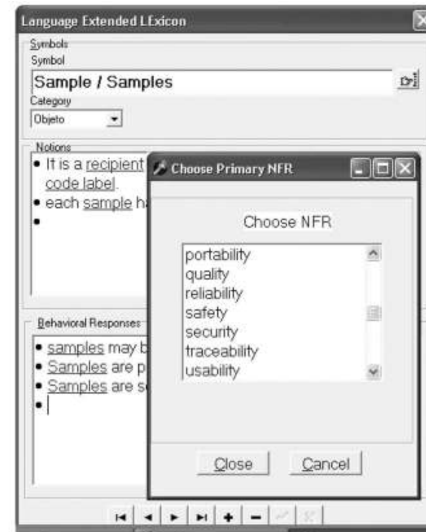


Fig. 4. Using the knowledge base to help eliciting NFR.

Take, for example, the case study performed within a clinical analysis laboratory (Case Study III). For each of the symbols represented in LEL, we asked ourselves and the customers what possible NFRs would have to be achieved so this symbol could be considered completely represented? During this process, we used a knowledge base implemented as a list of NFRs in the tool. Fig. 4 (screen on the left) shows the symbol Sample (for instance, fluids collected from a patient placed in a recipient), its notions, and its behavioral responses before we carried out an analysis for any NFR(s) that could be necessary to this symbol. Fig. 4 (screen on the right side) shows the use of the knowledge base entries. Fig. 5 shows the resultant symbol after we came to the conclusion that traceability was essential to that symbol. Traceability is essential because the laboratory could not afford to lose a sample. Drawing a new sample can be almost impossible sometimes or at least very painful. In Fig. 5, we can see a notion "Has NFR Traceability" indicating the addition of the NFR Traceability. Here, the use of the term Traceability can be understood in two different ways. The first means that some object or action in the real world must be traceable during a period of time. That is the kind of traceability that we referred to above. The second refers to the software engineering notion that one might be able to trace elements forward and backward throughout the several models used for developing a software system.

Now that we understand that a Sample has to be traceable, we must understand how this NFR might be achieved. We can use the knowledge base or we can ask

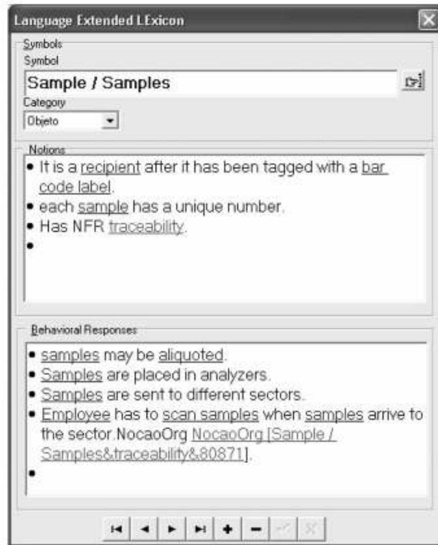


Fig. 5. Symbol after one NFR was picked up.

ourselves and the customers how we could guarantee this traceability would work. We may also apply both approaches, which in fact has been the most effective way to accomplish this task.

One of the responses we got from questioning about traceability in regards to the Sample entry was: "every time a sample is aliquoted (expression used in this domain that means to create an aliquot, or yet to draw from one recipient to another), this procedure has to be recorded so one can know which sample was originated from another sample." We represent this answer as an entry in the behavioral responses (LIS keeps a record of what sample is originated from another) of the symbol Aliquote sample (Fig. 6).

As we add a new behavioral response to the symbol Aliquote Sample, we have to create a dependency link in this behavioral response pointing to the NFR traceability stated in the notions of the symbol Sample. This link is represented using the pattern described above as: NocaoOrg [Sample/Samples&Traceability&80871]. This pattern shows that this behavioral response exists to satisfice[7] the Traceability NFR present in the symbol Sample. This pattern is used mainly by the OORNF tool to keep a rationale for the process. It can also be used as a quick guide to find out which NFR within a symbol has generated the need for a particular notion or behavioral response.

The other response we got was: "one should be able to know where a sample is now and where it has been before." We can see these two answers represented in Fig. 6, respectively, as a behavioral response in the Aliquote sample symbol (LIS keeps a record of what sample is originated from another) and another behavioral response in the Sample symbol itself (Employee has to scan samples every time a sample arrives to the sector).

Reasoning about the behavioral response we placed in the symbol Sample, we realized that this answer was not sufficient to satisfice the traceability NFR because we were not specifying how one could actually know where a sample is at any needed time. Again, using a traceability catalogue and asking ourselves how to solve this problem, we decided that in order to know the exact position of a sample at a time, it was necessary to scan this sample every time it is transported from one place to another. To represent this knowledge, we created a new symbol called Scan Sample that can be seen in Fig. 7. Although to scan sample may be at the end a functional requirements, it is in fact an operationalization for the NFR Traceability. We can also understand operationalizations as if they were, in fact, functional requirements that have arisen from the need to satisfice an NFR. This can explain why we frequently face doubts about if a requirement is functional or nonfunctional. The fact that an NFR when operationalized may result in new functional requirements points to the virtual impossibility of eliciting all the functional requirements firsthand.

This process is repeated for all LEL symbols; thus, at the end we have LEL expressing at least the basic necessary NFRs, and some of their operationalizations.

As said before, LEL is not the best tool to deal with dependencies among NFRs since they frequently involve many conflicts among possible solutions to satisfice one or more NFR. Thus, it is fair to say that at this point we have one first approach to satisfice NFRs, but it is still incomplete. It would be likely to have NFRs that we could not find operationalizations for, as well as undetected conflicts among these operationalizations, or among existing operationalizations. Therefore, we must perfect our knowledge of NFR satisficing for the domain using the NFR Framework with some slight adaptations.

3.2 Representing NFRs

3.2.1 The NFR Framework

Here, we use the same notion used by Mylopoulos [32]: that an NFR can rarely be said to be satisfied, that is, treating NFRs as goals we bring to bear the notion of partial satisfaction. This notion led Hebert Simon to coin the term "satisfice" [38]. Goal satisficing suggests that the solution used is expected to satisfy within acceptable limits. For instance, it expresses the idea that one can never say that a system is 100 percent safe against malicious offenders, but provides enough security measures to be considered secure.

As said before, the NFR Framework [32], [8], [10] views NFRs as goals that might conflict among each other. These goals are represented as softgoals to be satisfied. Each softgoal will be decomposed into subgoals represented by a graph structure inspired by the and/or trees used in problem solving. This process continues until the requirements engineer considers the softgoal satisfied (operationalized) [10], so these satisficing goals are understood as operationalizations of the NFR.

In accordance with [10], for us, an NFR has a *type*, which refers to a particular NFR, for example, security or traceability. It also has a subject matter or *topic*, for example, Sample as shown in the example in the previous section. We would then represent it as Traceability [Sample].

The NFR framework was extended to represent the operationalizations in two different ways. We called them

7. See Section 3.2.1.

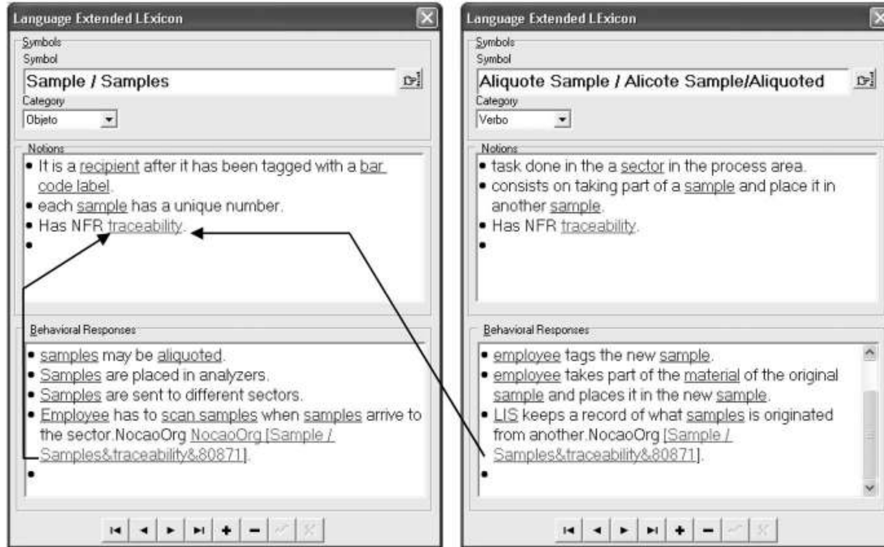


Fig. 6. Consequences of satisficing the NFR of the symbol sample.

dynamic and static operationalizations. Dynamic operationalizations are those that call for actions to be performed. Static operationalizations usually express the need for the use of some data in the design of the software to store information that is necessary for satisficing the NFR. Fig. 8 shows an example of an NFR graph where we can see these two types of operationalizations. Categorizing operationalizations as Dynamic or Static will later help with the direct mapping of these operationalizations into attributes or operations belonging to a class.

On the top of the Fig. 8 (extracted from the Case Study I —Light Control System), we can see the root of this graph "Safety [Room]." This root means that room is a place that has to be safe regarding illumination aspects, i.e., the room has to have enough light so people do not stumble and fall.

One of the operationalizations that represent part of the satisficing of this NFR can be seen on the left side of the figure represented by a bold circle denoting a static operationalization. Here, we can see the need for some information in the system that represents the minimum illumination in lux that can be used in a room.

Fig. 7. A symbol created to satisfice an NFR from another symbol.

On the bottom of Fig. 8, we can see dotted circles representing dynamic operationalizations. One of them, Safety [Room.Malfunction.User get informed], represents the requirement that a user be informed of any malfunction that occurs in the room. The letter S that appears inside each node represents that this subgoal is Satisficed. The letter P is used for those ones that are Partially satisficed or D for those ones that are Denied. A partially satisfied goal/subgoal means that not all of the possible alternatives to satisfice this NFR are being employed here, possibly due to conflicts with other NFRs.

It is important to stress that the identifier that appears close to the NFR on the root of the graph (NFR Topic) must be a LEL symbol. In Fig. 8, we see the root node Safety [Room]; therefore, room must be a LEL symbol. If one cannot find the word or sentence intended to be used as a topic for an NFR, then either one symbol represented in LEL has an alias not yet defined or LEL is incomplete and therefore, must be updated. If LEL is updated, all steps performed in the nonfunctional perspective must be revisited, e.g., look for possible NFRs for this symbol and search for possible operationalizations.

3.2.2 *Creating NFR Graphs*

To build the NFR model, one must go through every LEL symbol looking for notions that express the need for an NFR. For each NFR found, one must create an NFR graph where this NFR will be the root of the graph. This graph must be further decomposed to express all the operationalizations that are necessary to satisfice this NFR. This can be accomplished either using the knowledge base on NFRs or investigating what notions and behavioral responses were added to LEL to satisfice NFRs. These notions and behavioral responses will be candidates to be operationalizations for this NFR. These two approaches are not conflicting; actually, they are likely to be used together.

Since each NFR graph is derived from LEL symbols, we can model NFRs that spread over several symbols.

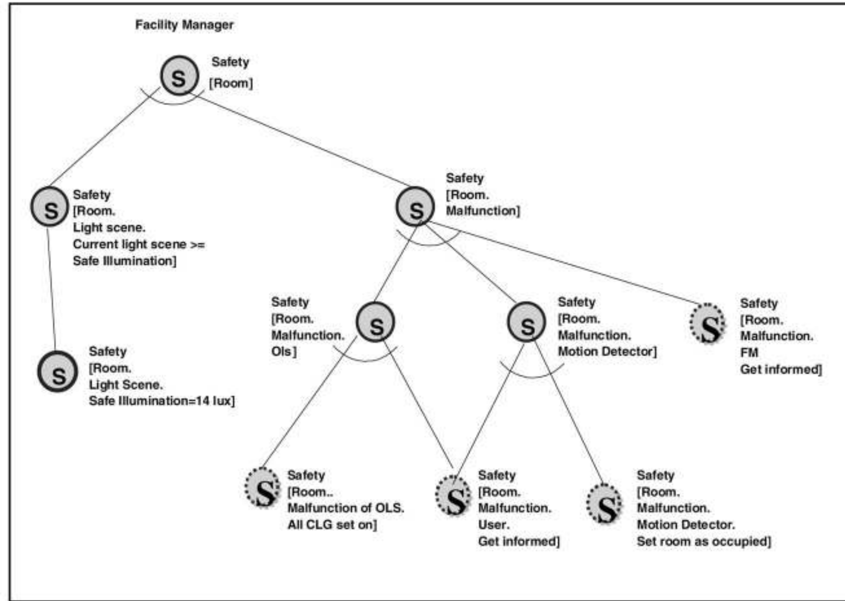


Fig. 8. An example of an NFR graph (from [13]).

Let us take for example the symbol Room belonging to a Light Control System (Case Study I). We can see a notion "Has NFR Safety" representing that the system must behave in a safe way regarding the amount of illumination in a room. Fig. 9 shows the entry for this symbol and illustrates how a NFR graph would be originated from there.

After we have represented the NFR graph root, we have to search for its operationalizations. Fig. 10a shows the result of this search; in this case the behavioral responses that were added to LEL to satisfice the NFR Safety[Room]. Using these behavioral responses we represent possible operationalizations for the Safety[Room] NFR as it can be seen in Fig. 10b. For instance, the third line "it can not establish less than a safe illumination" leads to the operationalization that is the leftmost one in Fig. 11. Once we had done that, we must now try to see what possible subgoals, if any, would represent an intermediary step between the graph root and its operationalizations.

We may proceed in two different ways:

1. Decomposing the root using a top-down approach. For example, we may choose to decompose the NFR Safety [Room] into a subgoal Safety [Room.Light Scene] representing that for a room to be safe, we have to assure that every possible light scene will be safe. We may also continue to decompose it to the point that the Current Light Scene will always be equal to or greater than the safe illumination.
2. We can continue the evaluation using a bottom-up approach. For example, take the operationalization (Safety [Malfunction of OLS.All CLG set on]) that expresses the need for having all the ceiling light groups (CLG) to be turned on in the occurrence of a malfunction of an outdoor light sensor (OLS), and the operationalization (Safety [Malfunction.User Get Informed]) that expresses the need for the user of a room to be informed of a malfunction. We could understand that these two operationalizations point out to an intermediary subgoal that decomposes the Safety [Room] NFR into a subgoal concerned with all types of malfunction (Safety [Room. Malfunction]).

The final result should be very similar, regardless of which method one chooses to use. Fig. 8 shows the final version of this graph.

After we have carried out this process for each LEL symbol, we will have a set of NFR graphs that will model the nonfunctional perspective. As such, we can now analyze all the graphs to check for possible conflicts and different design solutions that might then be negotiated with the customers.

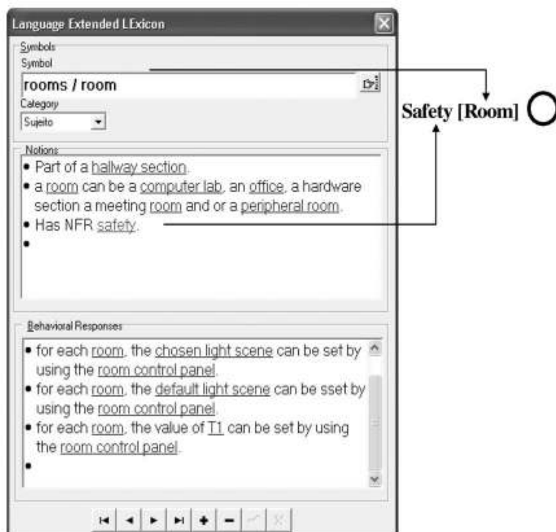
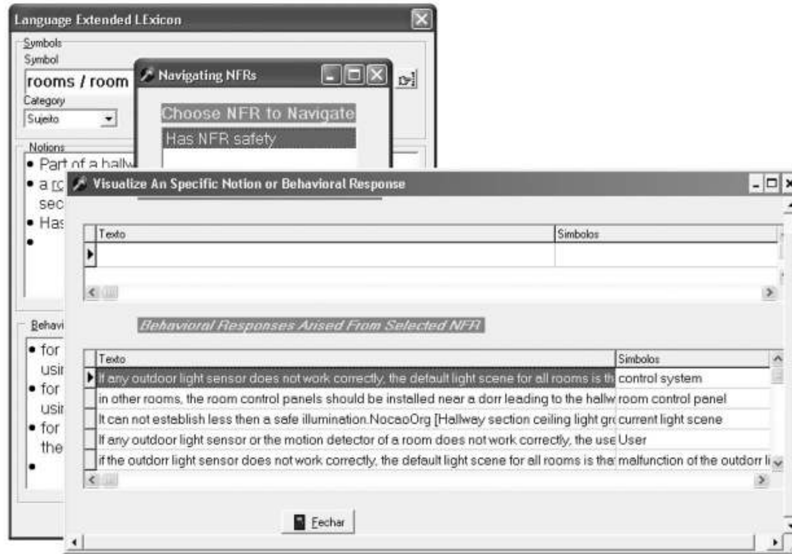
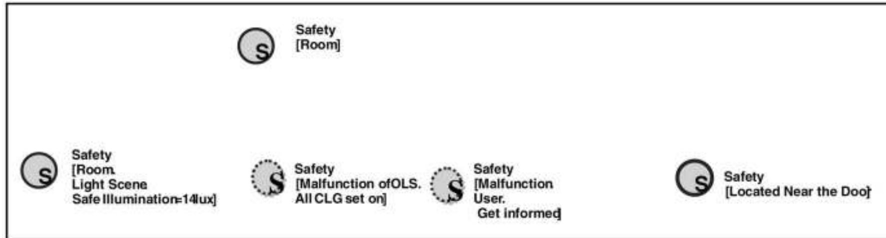


Fig. 9. Creating an NFR graph.



(a)



(b)

Fig. 10. (a) Navigating an NFR to find its operationalizations. (b) A first approach to decomposing an NFR.

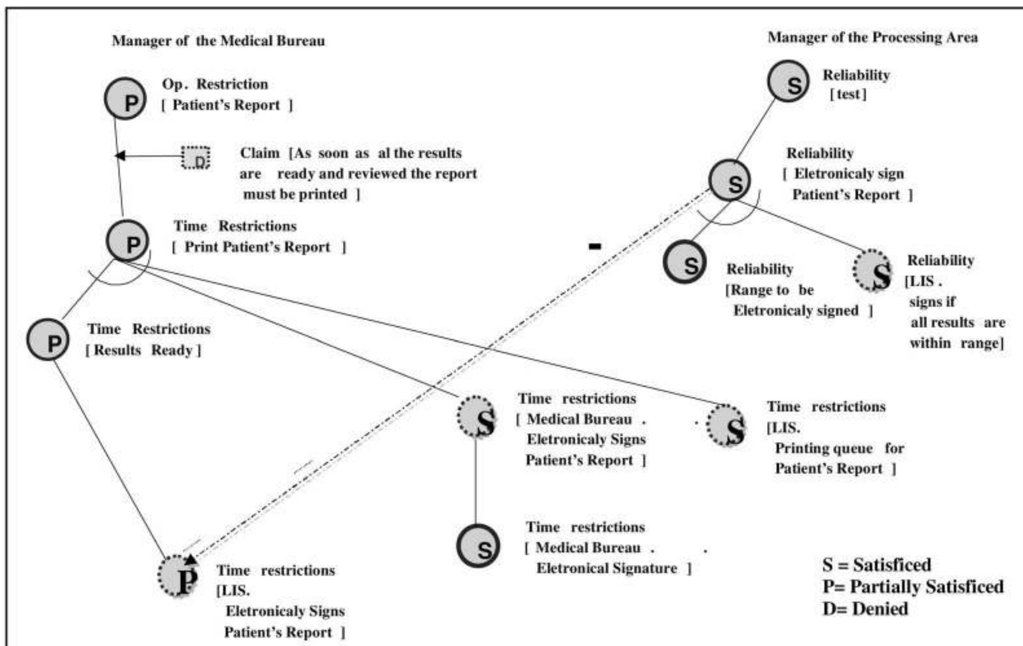


Fig. 11. Identifying and solving interdependencies among NFR graphs.

3.3 Identifying and Solving Conflicts

One important characteristic of NFRs is that we can often identify interdependencies among them, either positive or negative. In other words, an NFR may contribute positively or negatively to another NFR satisficing.

Once we have the set of NFR graphs, we have to search for possible interdependencies among NFRs. For example, an NFR pointing out that the software might need a high level of Security may have a negative impact (a negative interdependency) on another NFR like Usability. Frequently, when security is enhanced, usability aspects will be impacted. For example, enhancing security to internet banking might demand the user to create a secrete phrase that will be further entered every time the bank is accessed. While it might prevent unauthorized use of the account, it will also strongly affect the usability of the system since remembering and typing this phrase correctly may represent a burden to many users.

On the other hand, one might face an NFR for Performance that calls for saving space used for storing some information. To operationalize this NFR, one option could be to use compressed format. This operationalization will affect negatively another NFR that calls for a good response time when dealing with this information. An example of positive dependency could be the one for space performance versus privacy because to operationalize privacy one may choose to adopt some sort of cryptography mechanism that may lead to a more compressed format and, therefore, would positively contribute to the performance NFR. Further details can be seen in [10].

We propose three heuristics to help us with finding these interdependencies.

1. Compare all NFR graphs of the same type, searching for possible interdependencies. For example, we may put all the NFR graphs that have the type Safety together to see if there is any interdependency among them. Different customers may have different view points regarding safety concerns. Comparing them may facilitate the detection of conflicts.
2. Compare all the graphs that are classified in the knowledge base [34] as possibly conflicting NFRs. For example, compare graphs of Security with graphs of Usability. Comparing Graphs that are traditionally conflicting may disclose several conflicts.
3. Compare pair wise all the graphs that were not compared while applying the above heuristics. Pair wise comparison diminishes the importance of intuition or expertise to detect conflicts since it forces the comparisons that otherwise may be missed.

Fig. 11 shows an example of a negative interdependence found when we were applying the third heuristic during one of our case studies. This graph was extracted from a software system for clinical analysis laboratory (Case Study III).

This figure shows the pair wising of patient's report operational restrictions with reliability. Pair wising these graphs allowed us to see a negative interdependence from the subgoal that deals with the aspects of how to assure reliability when electronically signing patient's report, to the operationalizations that states that, in order to satisfice operational restrictions regarding time restrictions on printing patient's report, the system (LIS) should electronically sign all the patient's reports. This negative influence happens because the Manager of the Processing area said that in order to have reliable tests not all results can be signed electronically. Only the reports that have all test results within a predefined range can be electronically signed.

It is important to clarify that the subgoals that appear in Fig. 12a as partially satisfied (P) were considered satisfied (S) before we carried out the comparison. Only after we negotiated with the stakeholders to compromise with an intermediary approach were these subgoals considered partially satisficed. This is exactly what is represented in Fig. 12a, i.e., the patient's reports will be electronically signed by the system *only* when all the results are within a predefined range considered safe to this task.

Although being important, heuristic three may become inappropriate to be used when the number of NFR graphs is very large. Although we have not treated the scalability problem in detail, we understand that this kind of heuristic may need to be further studied addressing feasibility concerns [25]. We do not have a rule-of-thumb to this number but we have used this heuristic in a case study where we dealt with more than seventy NFR graphs and it was still worth using. Of course, automation will improve the effectiveness of the strategy; however, this is future work.

Once we have all the reasoning involved in NFRs' trade off done, we then have the nonfunctional perspective ready to be integrated into the functional perspective. It is necessary to understand where the conceptual models will be affected by the operationalizations elicited to meet the NFRs elicited. Furthermore, we need to reason how to include, if they are not yet represented, these operationalizations in the different conceptual models.

4 Integrating Nonfunctional and Functional Perspectives

Our strategy allows the integration process to be carried out in many different ways. The software engineer can integrate the NFRs into the use case, scenario and class models not necessarily in this order. The strategy also supports the integration of NFRs into class, sequence, and collaboration diagrams [37] being designed, or even into legacy systems.

Use cases [18], [37] and the scenario models [42] are frequently used for driving the construction of the class, sequence and collaboration diagrams. Therefore, if the software engineer integrates the NFRs into the use case and scenario models in the early stages of the software development process, the conceptual models that will arise later will naturally reflect the NFRs.

Doing so, the integration of the NFRs into class diagrams will turn out to be a process that will work more as a validation of the integration made before. Still, as diagrams at design level capture more detailed information than those used during earlier phases, some new designs

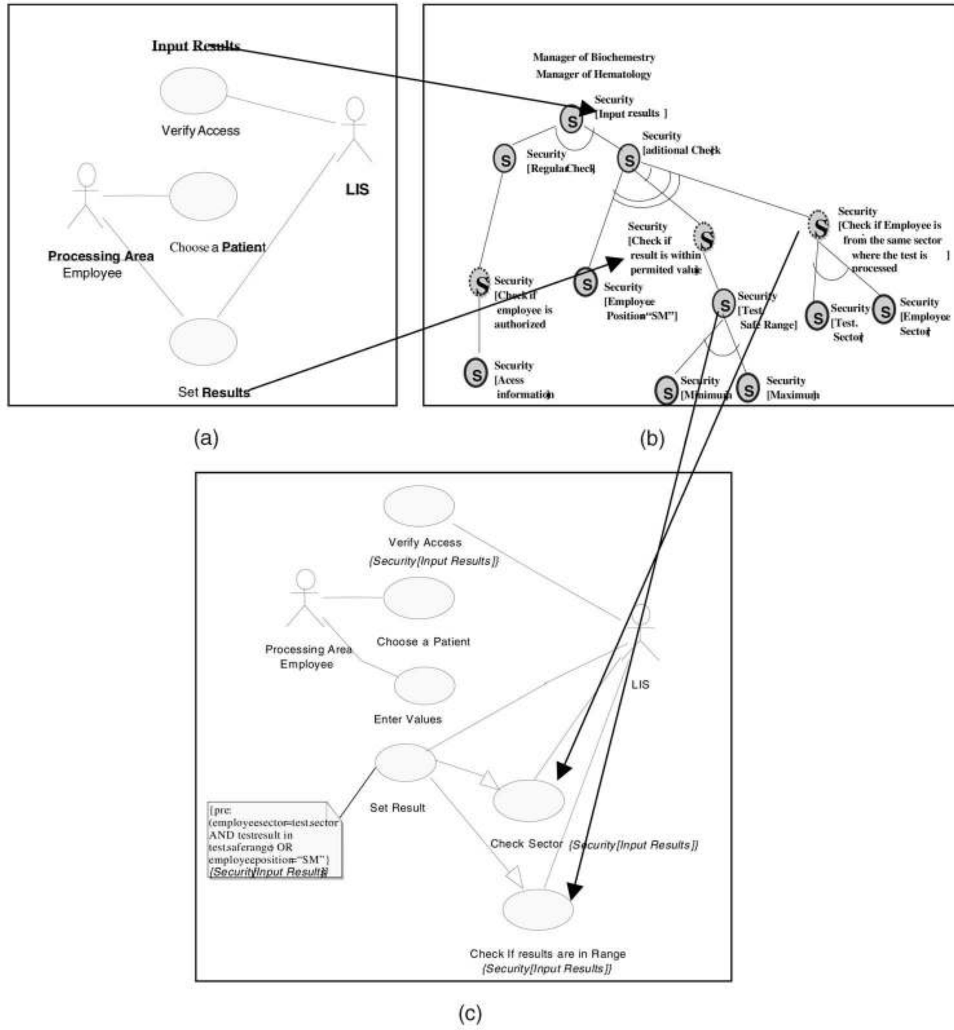


Fig. 12. (a) A Use Case before the integration process. (b) NFR graph to be intergrated. (c) Use case reflecting NFR.

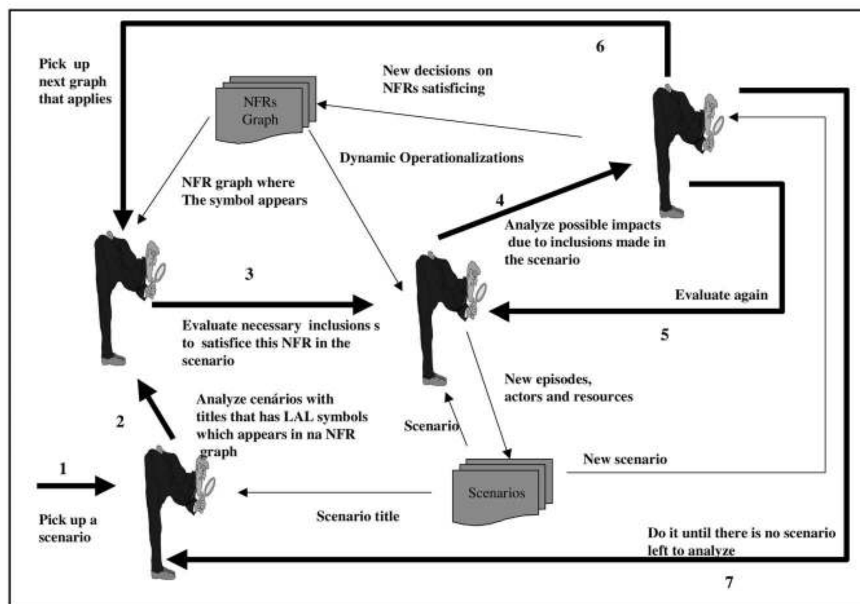


Fig. 13. The scenario integration process.

decision may happen here despite previous integration of NFRs into use cases and scenario models. Thus, integrating NFRs into the functional perspective during both early and later phases can be useful for reevaluating the integration under the viewpoint of new design decisions or requirements evolution.

4.1 Integrating NFRs into Use Cases

For every use case diagram in the functional perspective, we identify if any LEL symbol appears in the name of the diagram. We also identify if any LEL symbol appears in any of the use cases or actors of this diagram. For each LEL symbol we find, we will search the set of NFR graphs to identify those where this symbol appears. We may, eventually, find one or more NFR graphs that contain the symbol being searched for. We take every graph where the symbol appears and check if the use case diagram realizes the dynamic operationalizations in the graph, i.e., if there is any use case that does what is stated in the dynamic operationalizations.

LEL intends to capture every meaningful term used in the UofD. If a use case diagram does not have at least one LEL symbol, either there are symbols in LEL that may have aliases not yet specified or a symbol is missing. Most of all, all the actors in a use case diagram must be LEL symbols. If a symbol is missing, LEL must be updated and all the processes that are carried out within the nonfunctional perspective must take place again, e.g., search for NFRs that may apply to this symbol, create NFR graphs.

Every use case or actor included, due to NFR satisfaction, must be followed by an expression using the pattern: ***{NFR_Type[NFR_topic]}***. The use of this expression aims at adding traceability between the functional and nonfunctional perspectives.

The traceability between the functional and nonfunctional perspective played a very important role during the real life case study, most of all while reviewing the functional perspective models. NFRs may have a very vague nature and, in opposition to what happens in the case of functional requirements, it is unusual to have them clear in the software engineer's mind. This traceability link helps the software engineer while he is reviewing or changing the models. It helps, not only to see that some use cases are there to satisfy a specific need, but also to check during model evolution if any conflict has arisen because of these changes.

It may be necessary to specify some special conditions together with a use case, as for example, **pre** and **post** conditions needed to satisfice an NFR. These conditions may be specified beside the use case name between braces, preferably using OCL [35]. For example, in the clinical analysis laboratory domain (Case Study III), we faced one situation where satisficing a Security NFR applied to test results. We had to ensure that prior to setting any results to a patient's record, the system would have to check if this result is within a range considered safe, or if the employee inputting this result is a sector manager.

Fig. 12a shows a use case diagram used in the case study mentioned above. It portrays the diagram named "Input Results" before the integration process. This use case diagram expresses the set of use cases needed to input results to tests that were prescribed to a patient. In this figure, the actor LIS refers to Laboratory Information System.

According to the integration process, we identify LEL symbols in this diagram. The symbols are tagged as boldfaced and underlined words in the diagram. We now have to search the set of NFR graphs for the occurrences of any of these symbols. Fig. 12b portrays one of the NFR graphs we found. We picked out this graph because of the occurrence of the symbols Result, highlighted in Fig. 12a, as well as the occurrence of the symbol Input Result, which is the name of the diagram. Analyzing the use case diagram in Fig. 12a, we can see that regular access validation was already satisfied in the use case diagram, while the other two had to be added. Fig. 12c shows the same use case diagram after adding the necessary use cases and special conditions to the diagram. We can see in Fig. 12c that we included two new use cases that are used by the use case "set result." They were included as an "include" link since they will be used by many other use cases. Both use cases came from the dynamic operationalization in the graph portrait in Fig. 12b.

These operationalizations state the need for checking if the result is within a range considered secure. They also show that it is necessary to check if the employee who is inputting the result belongs to the same sector where the test is being processed. These two checks can be ignored if the employee holds a position as a sector manager (represented by "SM"). The use case "Set result" is the one responsible for inputting the result to the patient's report; the condition mentioned above is linked to this use case as a note. This note specifies that before inputting any results the LIS has to check that these conditions are met.

This process has to be carried out for all the use case diagrams that compose the software specification.

4.2 Integrating NFRs into Scenarios

We also propose to integrate the NFRs from the nonfunctional perspective into scenario models. Here, we view scenarios as an artifact for requirements elicitation that is not necessarily linked to any use case, although the integration process is still valid to be used when scenarios are viewed as a refinement of a use case.

We use the scenario description proposed by Leite [28] to illustrate the process. On more time, the integration process will be anchored in the use of LEL symbols. Each existing scenario will be evaluated, searching for LEL symbols being used in the title of the scenario, in the resources description, in the actor's description, or in the goal description. For each symbol found, we search the set of NFR graphs looking for this symbol. Once we find one or more NFR graphs where the symbol appears, we have to check if all the operationalizations stated in each graph are already satisfied in the scenario description. If it is not, we have to update the scenario so that this condition holds. There is no prescribed order on how to analyze a scenario. This process continues until all the scenarios have been analyzed. Fig. 13 illustrates the process. Dynamic operationalizations will probably be satisfied by one or more episodes while static ones will be part of one or more episodes.

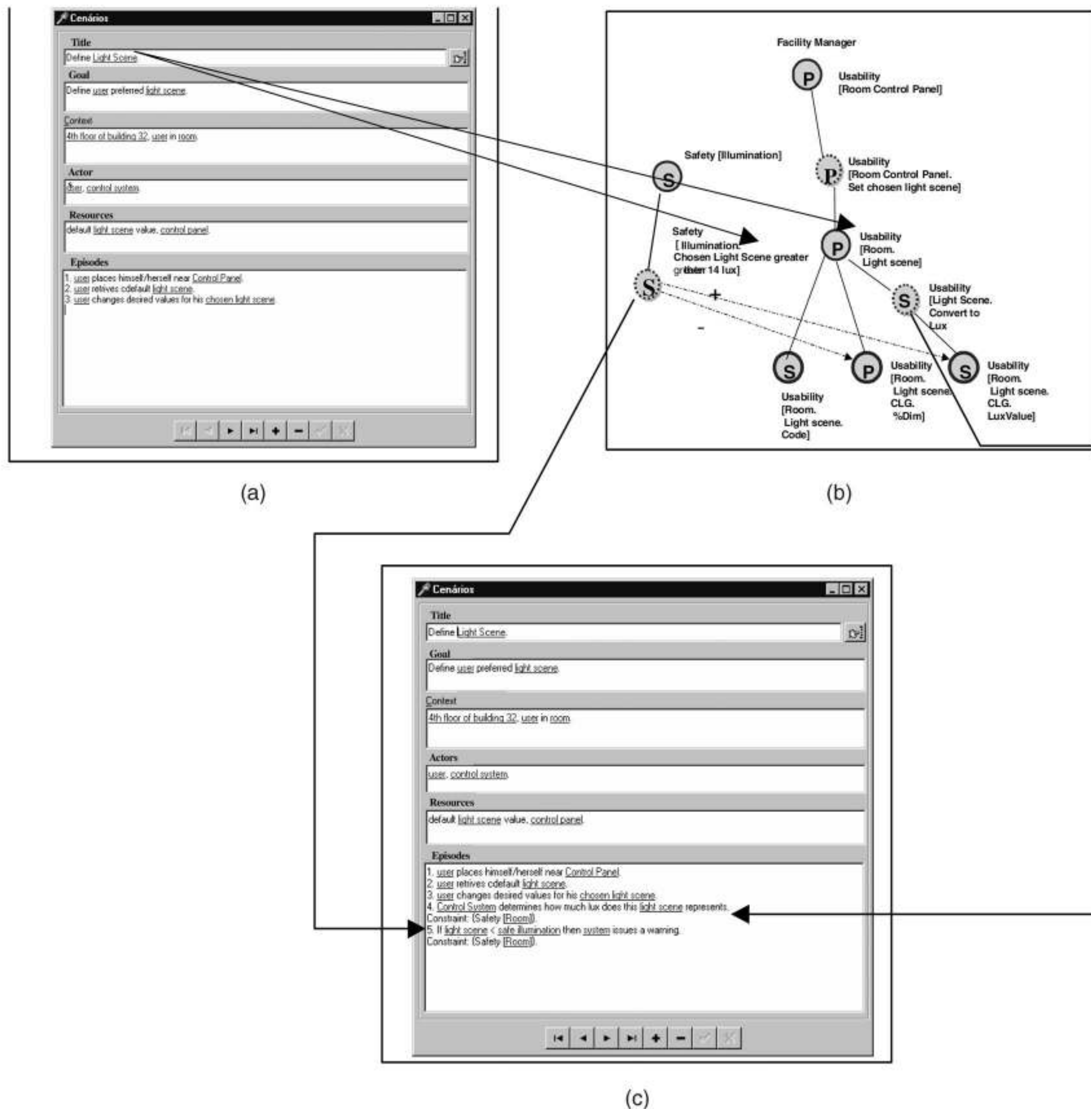


Fig. 14. (a) Scenario from a light control system. (b) Part of the NFR graph containing the symbol "light scene." (c) The resulting scenario.

As we did when dealing with use cases, the title for a scenario has to contain at least one LEL symbol. If one can not find a LEL symbol in the scenario title, either a synonym is missing or LEL is incomplete and should therefore be updated. Note that, when updating LEL, one may go over the nonfunctional perspective again, i.e., evaluate the symbol for possible NFRs, represent in the graphs, etc.

The same way as in use cases, the information included in a scenario to satisfice an NFR must be followed by the expression: *Constraint: {NFR_Type[NFR_topic]}*. We use this expression in the same way we use it in the use case diagrams. It aims at adding traceability between the functional and nonfunctional perspectives.

This process was used in Case Study I (detailed on Section 5), regarding a light control system, to integrate the NFRs into the scenarios. An example is shown in Fig. 14a.

All the expressions underlined in the scenario description are LEL symbols. Following the process for each symbol we found in the scenario, we searched every NFR graph looking for the occurrence of these symbols. Take, for instance, the symbol "Light Scene," Fig. 14b portrays part of the two graphs we found. As it can be seen there, one of the graphs illustrates that, in order to have usability in the system, we must have a control panel that can be used for setting a desired light scene.

To establish a light scene, the user will set how much the light must be dimmed. This information is, by default, passed in a percentage form. In Fig. 14b, we can see that a graph illustrating that in order to satisfice the NFR Safe[Illumination], we have to store not only the percentage value that represents how much you are dimming the lights, but also the equivalent in lux. This is done because the system has to check whether the user is inputting a safe value for the light scene or not. This contributes to satisfice another graph that states that any light scene should be greater than 14 lux. Analyzing the scenario in Fig. 14a, we can see that this scenario is not in conformance with what is stated in the NFR graph. Thus, we had to add two new episodes to this scenario to satisfice the NFR. Fig. 14c shows the scenario after the integration process. This same process was then carried out for all the other scenarios available.

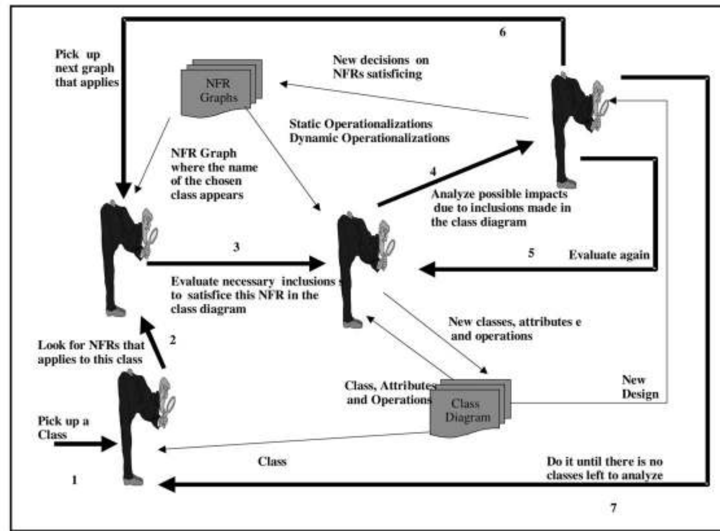


Fig. 15. The class diagram integration process.

4.3 Integrating NFRs into Class Diagrams

As in the previous models, integration of the nonfunctional perspective into the class model will be based on the use of LEL. Here, it means that every class belonging to the class diagram has to be named using a LEL symbol. The use of LEL as an anchor to construct both perspectives will facilitate their integration. It can also be used for validating both models since, if for some reason one cannot find a LEL symbol for naming a class, it means that either any LEL symbol has an alias that was not yet considered, or the symbol is missing in LEL definition and, therefore, should be added to it. If that is the case, one may go over the nonfunctional perspective again, i.e., evaluate the symbol for possible NFRs, represented in the graphs, etc.

Using this anchor, the integration process is centered on searching for a symbol that appears in both models, and evaluating the impacts of adding the NFR's operationalizations to the class diagram. Fig. 15 depicts the integration method for the class diagram. We start the process by picking out a class from the class diagram. There is no order for choosing one class or another. We search all the NFR graphs looking for any occurrence of this symbol. For each graph where the name of the class we are searching for appears, we have to identify the dynamic and static operationalizations from this graph.

For dynamic operationalizations found, we have to check if the operations that belong to this class already fulfill the needs expressed in the graph's operationalizations. On the other hand, for static operationalizations we have to check if the class attributes already fulfill the needs expressed in the graph's operationalizations. If they do not, we have to add operations and attributes to the class. Note that, adding new operations may sometimes call for the inclusion of new attributes in order to implement the desired operation or vice-versa.

Let us take for example a class named *Room* from a light control system (extracted from Case Study I detailed in Section 5). We had to search all the NFR graphs in the nonfunctional perspective looking for the symbol Room. One of the NFR graphs we found is shown in Fig. 8, where we can see five operationalizations, four dynamic and one static. The four dynamic operationalizations state that the software must:

1. Turn all the lights on when a malfunction of the outdoor light sensor occurs.
2. Advise the user of a malfunction of both outdoor light sensor or motion sensor.
3. Set the room as occupied in the case of a motion sensor malfunction.
4. Advise the facility manager (FM) of any malfunction.

We have to check if any operation in the class *Room* already performs these actions. If they do not, we should add operations to handle these actions. On the other hand, the static operationalization states that there should be an attribute fixing the minimum amount of light in a room as 14 lux. In this case, we have to check if there is an attribute in the class *Room* to store this information.

Notice that, if the class that we are analyzing is part of a generalization association, we have to check if any superclass or subclass of this class does not have operations, or attributes that satisfy the needed operationalizations.

4.3.1 Heuristics on How to Use UML Class Diagrams to Handle NFR

To integrate NFRs into class diagrams calls for some extensions to be made on how to use UML notation for these diagrams. Below, we present four heuristics on how to proceed.

1. Classes created to satisfice an NFR may have the name of the class followed by a traceability link that points out to the NFR whose operationalizations demanded the class be created. This link will follow the syntax: ***{NFR [LEL symbol]}***. Since NFRs are often more difficult to be on designers' minds than functional requirements, having this traceability link avoids classes to be withdrawn from the class diagram during a reviewing process, because one could not find any reason why this class must exist.

Fig. 16. Example of a class created to satisfice an NFR.

Fig. 16 shows an example of such a class. This class was created when we were analyzing the class *Control Panel* in the class diagram for a light control system (Case Study II described in Section 5). We searched the set of NFR graphs in the nonfunctional perspective looking for the symbol Control Panel.

One of the graphs we found is showed in Fig. 17. We can see in this figure that this graph calls for two dynamic operationalizations: 1) Set Password and 2) Ask Password. These two operationalizations were necessary because only the facility manager can change some parameters of the system for security reasons. Therefore, there has to be not only a control panel for the facility manager, but also a process to check the password in this control panel to avoid unauthorized people using it.

Since we did not find in the class *Control Panel* anything that would satisfice this NFR, we decided to create a new class (*FMCP*-Facility Manager Control Panel), as a subclass of the class *Control Panel*.

It is important to make it clear that the creation of a new class to satisfice an NFR will always be a design decision. The software engineer could have chosen, in this case, to add the same attributes and operations present in the class shown in Fig. 16 to another already existing class such as the *Control Panel* class.

2. Adjacent to each operation that has been included to satisfice an NFR, we add a link to the nonfunctional perspective. As in heuristic one, this is to enforce traceability between models, so the designer can easily check nonfunctional aspects whenever he changes anything in this class. The link will follow the same pattern as in heuristic one.

   Let us take for example the class *Room* mentioned before. Suppose we add an operation named AdviseUserofMalfuntion() in order to perform one of the operationalizations, we should then represent it as follows: AdviseUserofMalfunction() {Safety [Room]}.

   We present here a different syntax from the one presented in [14] for NFRs traceability. We have changed it so one can trace exactly which NFR has originated the need for a specific operation. This traceability was not possible in the former work when there were more then one NFR per class. This also holds for heuristics 3 and 4.

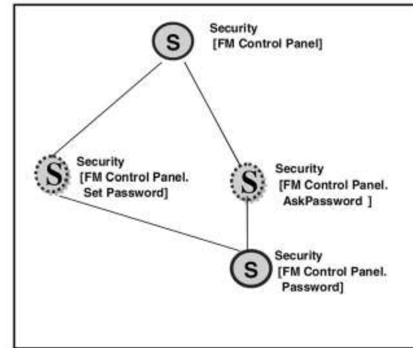


Fig. 17. NFR graph for a light control system.

3. If an NFR calls for pre or post conditions to apply for an operation, we may add these pre or postconditions to the respective operations.

   This heuristic is used for dealing with operational restrictions that some NFRs impose. These operational restrictions should be inputted as pre or postconditions to an operation and whenever possible should be stated using OCL [35]. These pre and postconditions can also be stated in a note linked to the class.
4. Adjacent to each attribute that has been added to satisfice an NFR we may use the same expression we use in the operations to establish a link to the nonfunctional perspective.

Fig. 18b shows an example with the results of applying the heuristics two to four. This figure shows the class *LightGroup* after the integration process during the Case Study II, detailed in Section 5. Fig. 18a shows the class *LightGroup* before we carried out the integration process. During the integration process, we analyzed each class we had in the class diagram. When we picked out the class *LightGroup*, we searched the nonfunctional perspective looking for any occurrences of this symbol. Fig. 18b illustrates one of the graphs found. This graph relates to the NFR Safety needed when lights are dimmed. One of the nodes (subgoals) that decomposes this graph, shows that in order to be able to criticize if the illumination is greater than 14 lux, the light group has to be able to calculate the equivalent in lux to the percentage set to dim the lights (originally considered). Since there were no attributes or operations in the class doing that task, we added the attribute LuxValue and the operation CalculateLuxValue(). Another graph found was the one regarding the NFR Safety applied to the Control System that can be seen in Fig. 18d. As we can see in Fig. 18c, to satisfice this NFR there has to be an operation to dim the lights to 100 percent, to be executed if the light group does not receive a signal from the control system after T4 sec.

Again, there were no operations in the *LightGroup* class to perform this task, thus we added the operations ListenPulse() and DimLights100%() together with the attribute LastPulse necessary to implement the ListenPulse() operation. Notice that beside the operation DimLights100%(), we see a precondition that states that this

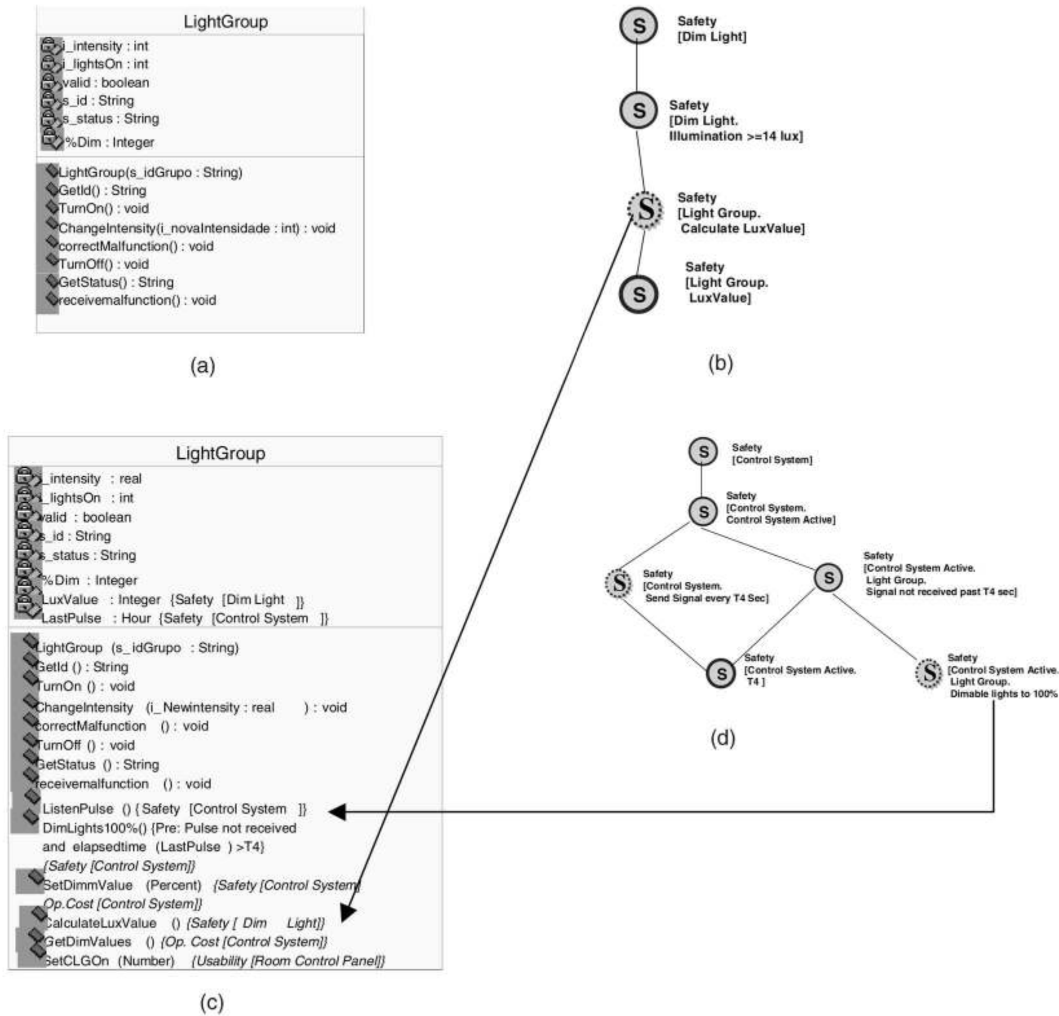


Fig. 18. (a) Class LightGroup before NFR integration. (b) NFR graph for safety applied to Dimm Lights. (c) Class LightGroup after the integration process. (d) NFR graph for safety applied to the Control System.

operation will be executed only if this class does not receive the signal within T4 sec.

Another example can be seen in Figs. 19a, 19b, 19c, 20a, and 20b. This example was drawn from the third case study regarding a software system for a clinical analysis laboratory.

Fig. 19a shows the class *Result to Inspect* before the NFR's integration. This class represents all the results from tests that come from one or more analyzers (special equipment that performs several tests automatically) and are waiting to be inspected by specialized employees. These employees will analyze if the results are, or are not consistent with other results and/or patient's history.

Using the integration process, we searched every NFR graph looking for any occurrences of this LEL symbol. Fig. 19b shows one of the NFR graphs found. We can see in Fig. 19b that there are some specific concerns about security regarding the results to inspect:

1. The system must perform a regular check to see if the employee is authorized to access the software module that implements the input of result to inspect.
2. The system may check if the employee is trying to input or change results belongs to the same sector where the test is processed.
3. The system must check if the inputted value is within a range considered safe to be inputted by any regular employee. Values out of this range can only be inputted by the sector manager.

Fig. 19c shows the class after the integration process. We can see in this figure that three new operations were added to satisfice the required operationalizations found in the NFR graph shown in Fig. 19b. We can also see a note with the pre and postconditions resulting from the integration. Note that, in this case, three existing operations were affected by pre and post condition. We use the same pattern for traceability to establish which NFRs these conditions came from. We do not need to do that for the operations that were driven by NFRs, such as CheckresultinRange, since the operation already carries the traceability link. However, using it or not is a choice for the software engineer.

Let us take for example the operation SetResult(). This operation is responsible for associating a result with a test in

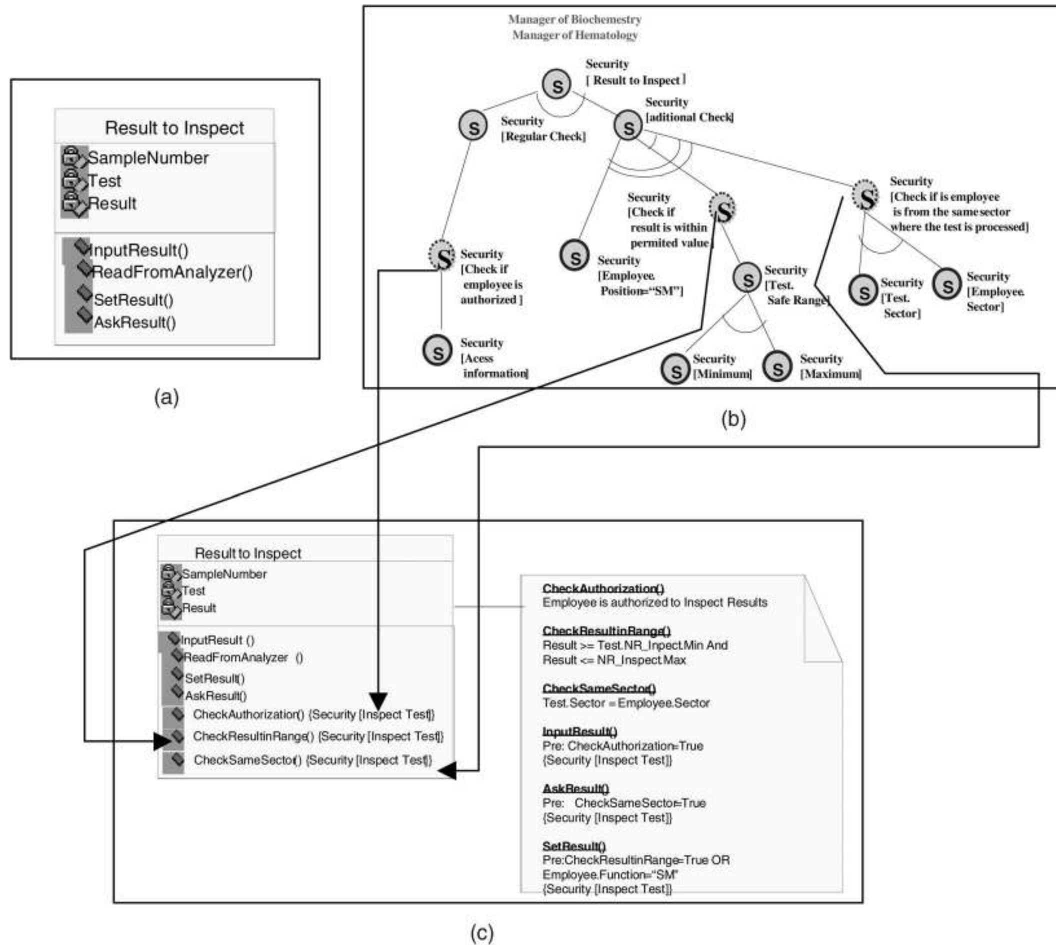


Fig. 19. (a) Class result to inspect before NFR integration. (b) A first NFR graph found to be integrated. (c) class result to inspect after integrating the first NFR graph.

a definitive way. We can see in Fig. 19c that this operation can only be executed if the result was checked to be in range or if the employee inputting the result is a sector manager.

Continuing the process, we found another NFR graph with the symbol Result to Inspect. This graph is portrayed in Fig. 20a. In this graph, we can see that in order to satisfice the NFR Reliability applied to Result to Inspect, the system must tag results that are out of the range considered to be normal, i.e., results that are out of the range usually experienced by the average population. The software must also allow the employee to tag some tests to be repeated, as well as to provide a way to send all these tests to the analyzer. Examining the class *Result to inspect*, we could see that none of these conditions were satisfied yet and, therefore, we added three more operations and two attributes. Fig. 20b shows the final design of the class *Result to Inspect*.

Once all the classes have been used for searching the graphs from the nonfunctional perspective, the class diagram will then reflect all the attributes and operations that are necessary to implement the needs arisen from NFRs satisficing.

It is important to note that not all LEL entries are directly matched in the functional models. However, there are LEL entries that will be connected to the functional model in an indirect way, i.e., an LEL entry that is a direct match will have behavioral responses or notions that will be considered in the integration. It occurs that in those behavioral responses or notions we may have, by the principle of circularity, an LEL entry that applies to the matched entity and to others as well. For example, if we have an entry that is a verb or a verbal phrase, this entry could be present in more than one class of the functional model, as an operation. Aside from that, it is very likely that this entry/operation will be present as a use case or part of a scenario; thus, when integrating NFRs to these models, we would end up dealing with the NFRs needed for this entry and later propagating them to the other models.

Another example is the case of the NFR safety. We have seen it applied to the class Room, since it was present in the symbol Room in our LEL, but the NFR safety could also appear in other LEL entries, thus being spread over the functional model.

Last, we need to integrate the NFRs into the sequence and collaboration diagrams.

4.4 Integrating NFRs into Sequence and Collaboration Diagrams

Integrating NFRs into the sequence and collaboration diagrams is done by examining every class of the class

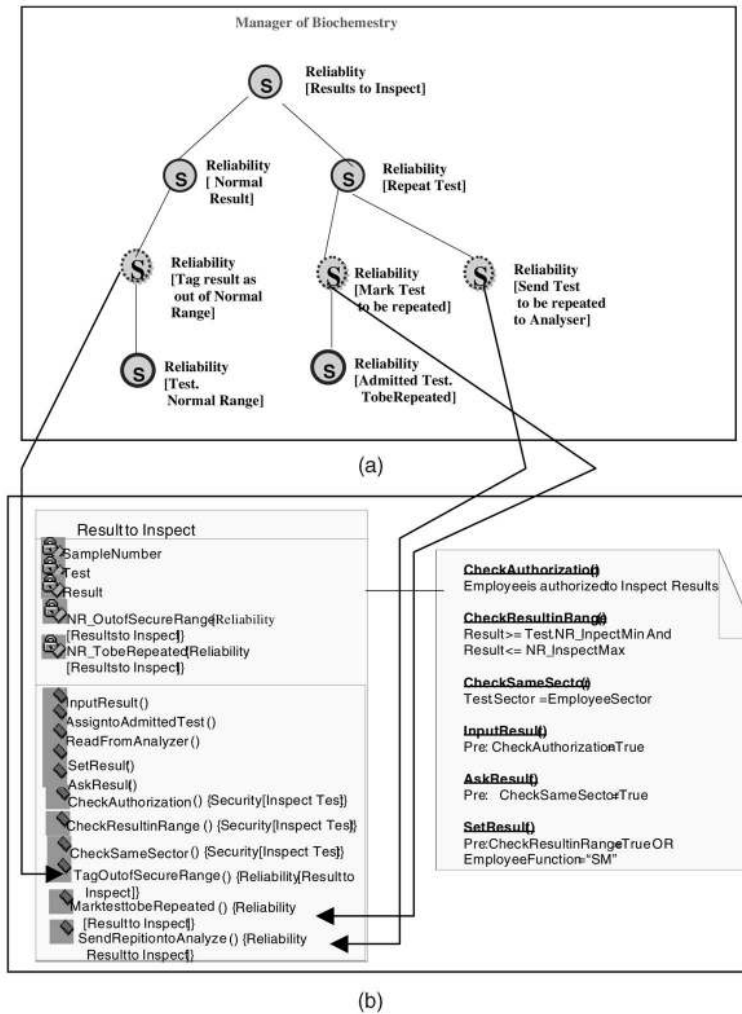


Fig. 20. (a) Another NFR graph. (b) Final design for the class result to inspect.

diagram. For every operation included because of NFR satisfaction, we may search the sequence and collaboration diagrams where this class appears. For each diagram we find, we must check if the new operations added due to NFR satisfaction will imply any change in this sequence or collaboration diagram.

It may be necessary to add classes, messages, or both to the diagram. If there is any pre or postcondition attached to an operation, we may also need to specify it attached as a note to a message.

We must be able to represent that new messages together with pre and post conditions in the sequence and collaboration diagrams were added due to NFR satisficing. This is done by using a note linked to the message where the condition will apply. This note will contain the expression that portrays the pre or postcondition. Any message included in these diagrams due to NFR satisfaction will have the same traceability expression we used with attributes and operations.

Let us take for example the class Result to inspect shown in Fig. 20b. Applying the strategy, we searched the collaboration diagrams seeking for instances of the above class. Fig. 21 shows the one we found.

Examining the existing diagram, and the operations included in the class due to NFR satisficing, along with the special conditions represented in the note, we concluded that some changes had to be made in this diagram. Fig. 22 shows the resulting diagram. We can see in this figure that the message tagged with the number 6 was added to satisfice a Security NFR regarding Input Results. This was necessary so the software could check, prior to allowing any

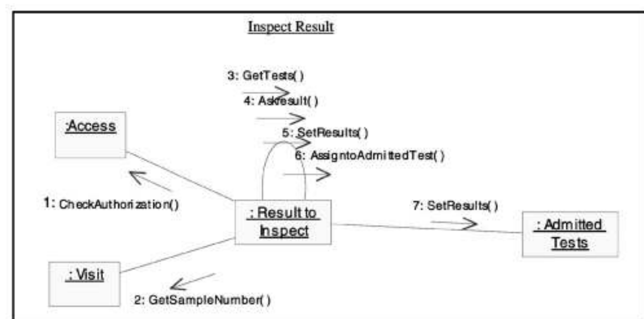


Fig. 21. Collaboration diagram where the class ResulttoInspect appears.

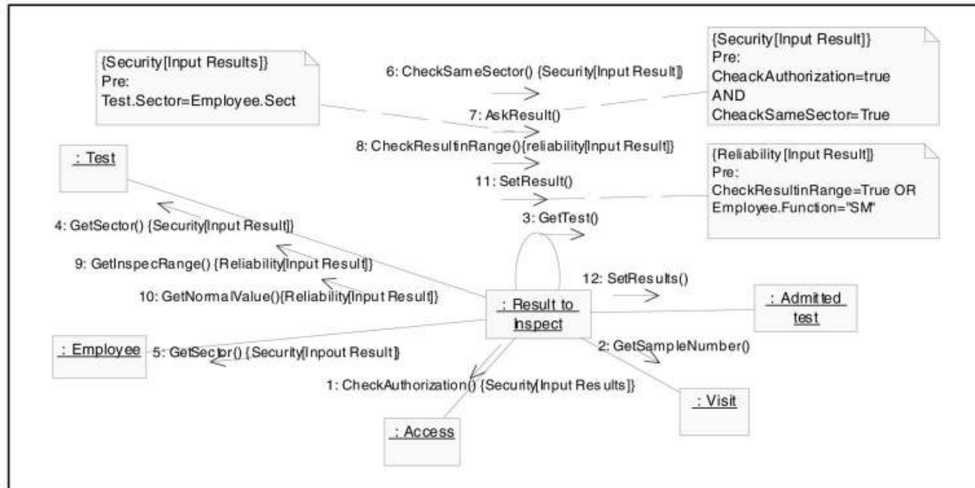


Fig. 22. Resultant collaboration diagram.

results to be assigned to any of the admitted tests, if the employee inputting the result works in the same sector where the test is performed. For the same reason, we can see a precondition added to message 7. This precondition establishes that before asking for any results the system has to check if the employee belongs to the same sector where the test is processed.

Another example that can be seen is the operation tagged with the number 8. This operation was added to the collaboration diagram to satisfice NFR Reliability for the class Input Result (See Fig. 20b).

Due to space limitation, examples using sequence diagrams and further comments about the other occurrences will be omitted.

5 Using the Strategy

In order to gain confidence in our strategy, we performed three case studies. All the case studies were inspired by the project replication strategy idea as proposed by Basili [1]. Since our strategy can be seen as an addition to most of the software development processes, we used three different projects, each one using its own independent team for developing a conceptual model for the software.

These conceptual models aimed to represent the clients' requirements under the viewpoint of each one of these three teams. On the other hand, the integration team, represented by one of the authors, was in charge of evaluating how well these conceptual models were expressing NFRs.

As a result of applying the process, we found several new classes, operations, and attributes that should have been specified in the conceptual model and were not. These classes, operations, and attributes were understood as errors in the conceptual models, since, if the software system driven from these conceptual models had been delivered to the client without these new classes, operations, and attributes, the lack of them would have been pointed out by the client as errors in the software that would have to be fixed.

A qualitative evaluation of the changes performed in the software, due to NFR satisficing, would be highly subjective, thus, we decided to perform a quantitative analysis regarding how many new classes, operations, and attributes were added to the conceptual models to satisfice NFRs. We understand that these numbers would roughly represent the amount of effort, time, and money that would have been spent to change the software system to comply with these requirements.

It can be stated that including new classes, operations, and attributes may lead to new errors, as well as to a not so high quality model regarding aspects as coupling and cohesion. Although it is true, we think that, first, not all of the changes may introduce new errors. Second, coupling and cohesion characteristics could be easily achieved after the introduction of the new classes, operations, and attributes. Third, not having these new classes, operations, and attributes would be, undoubtedly, errors in the software system.

Case Study I was conducted using the conceptual models created by Breitman [6] as part of her PhD thesis. It is important to mention that Breitman has a background in NFR [5]. She carried out a case study using the implementation of the Light Control System for the University of Kaiserslautern. This system was first proposed in a workshop on Requirements Engineering at Dagstuhl in 1998 [4]. We used the system specification distributed during the Dagstuhl Seminar, including LEL definition enclosed in the original problem specification, to build the nonfunctional perspective. Once we finished it, we integrated the NFRs found in this perspective with the conceptual models that were built by Breitman. The new classes, operations, and attributes that have arisen from this integration were counted as errors in the original conceptual model.

Case Study II was conducted using the conceptual models created by a group of graduate students of PUC-Rio during a software project course. They also used the specification distributed during the Dagstuhl Seminar [4] to build their conceptual models.

TABLE 1
Results from the Case Studies

	Existent Class	New Classes	%	Classes Affected	%	Existent Operations	Operations Found	%	Existent Attributes	Attributes Found	%
Case Study I	8	2	25	6	75	105	50	48	22	10	45
Case Study II	15	3	20	4	27	115	45	39	32	8	25
Case Study III	39	9	23	19	48	213	78	36	105	31	29

As the source of requirements for both Case Studies I and II was the same, in Case Study II we used the same set of NFR graphs from the nonfunctional perspective in the Case Study I and integrated them to the conceptual models that were built by the students. Again, new classes, operations, and attributes that have arisen from this integration were counted as errors on the original conceptual model.

Case Study III was conducted together with a software house that is specialized in building software for clinical analysis laboratories. They were responsible for developing a new information system for a laboratory. They built a conceptual model expressing the software requirements for the laboratory. LEL (previously constructed by one of the authors) was used for naming the classes used in the class diagram.

One of the authors of this paper acted as the second team in this case study and built the nonfunctional perspective by making use of structured and open-ended interviews with some of the customers and by using some available documentation such as ISO 9000 quality manuals. Observation and Protocol analysis techniques [19] were also used, although in a lower scale. The other team was composed of two Senior Analysts (with an average of 10 years experience) and one Junior Analyst with three years experience.

Once the nonfunctional perspective was ready, we used the integration process showed in Section 4 and, consistent with the other case studies, all the new classes, operations, and attributes that have arisen from this integration were counted as errors in the original conceptual model.

In order to measure the overhead of using the proposed strategy, we first measured the time spent by the software house team (Case Study III) from the initial phases of the software development until the conceptual models were finished.

We recorded a total of 1,728 man hours of work. On the other hand, we measured the time we took to build the nonfunctional perspective and integrated it with the conceptual models. We consumed 121 man hours to do that. Therefore, we state that the estimated overhead was 7 percent. This number is also coherent with the overhead found (10 percent) in previous case studies of [14]. The difference can result from either some minor mistake in the measurement, or improvements introduced by the new strategy.

Take for example the numbers from Case Study III in Table 1. We found nine new classes, where before, there were 39 due to the use of our strategy. It is reasonable to say that these classes would be potentially ignored when implementing the software system and, therefore, would represent an additional burden either during acceptance tests or after deployment.

All three case studies presented a considerable number of changes in the analyzed conceptual models, as seen in Table 1.

Compiling the numbers, we can see that 46 percent of the existing classes were somehow changed to satisfice NFRs. We can also see that we found a number of new operations that represented 39 percent of the existing ones.

Similar numbers could be found for attributes. We found a number of new attributes that corresponded to 30 percent of the existing attributes. These numbers clearly suggest that, if the strategy had been used during the development of the evaluated software, we could have got a more complete software specification, and probably, fewer demands for changes after deployment. These numbers are consistent with the numbers we got in previous case studies that have used a former version of the strategy [14]. During this case study, a third team was participating without interacting with the author and, thus, paying little attention to NFRs. This team was responsible for developing the software for the administrative/financial area of the laboratory.

We measured the total effort spent from the requirements elicitation to the moment prior to software deployment for both teams. Here, we considered the author as part of one of the teams. Therefore, all the time spent to integrate the NFRs was counted as time spent by the team in charge of developing the software for the processing area.

The team in charge of the software for the processing area spent 6,912 man hours from requirements elicitation to software deployment, in opposition to the 8,017 man hours spent by the other team.

The first team was responsible for 52 percent of all the coding (measured in numbers of lines of code inside the programs) while the second team was responsible for 48 percent of all coding. Therefore, we should expect to have a similar number of hours for both teams to deploy software. However, we ended up having the third team spending 15 percent more time to deploy the system. We believe that, consistent with the numbers presented in [14], the reason for the difference of time spent by each team was due to the effort spent on fixing problems, during the test and acceptance phases, due to NFRs not considered before.

Unfortunately, we do not have the time that teams from Case Studies II and I took to build the conceptual models. We do have the time we spent to build the nonfunctional perspective and integrate it to the conceptual models.

Quoting Ian Alexander, in a recent participation in the Requirements Discussion List:[8] "The problem with RE statistics is that we don't have a scientific set of double-blind-controlled experimental results. Running even two identical projects under controlled conditions with just one variable different doesn't sound easy; running 20 would be a nightmare." Hence, in spite of the many variables involved in the case studies, we understand that since all the three models were significantly improved after we applied our strategy, even those modeled by a participant with an NFR background, suggests that the strategy clearly brought benefits to the resulting conceptual models.

6 Conclusion

Errors due to NFRs are the most expensive and difficult to correct [7], [15], [17], not dealing with, or improperly dealing with them can lead to more expensive software and a longer time-to-the market. However, most of the software engineering methods do not deal with NFRs in an explicit manner. A survey [20], from a small sample of organizations, of the state of the practice in terms of nonfunctional requirements has shown that:

1. nonfunctional are often overlooked,
2. questioning users is insufficient,
3. methods do not help the elicitation of nonfunctional requirements, and
4. there is a lack of consensus about the meaning and utility of nonfunctional requirements.

Only recently, research results are showing ways of dealing with NFRs [10], [14], [40] at the software definition level.

Our contribution fills this gap in software development. We presented a strategy that tackles the problem of NFRs elicitation and proposes a systematic process to assure that the conceptual models will satisfice these NFRs. The strategy is based on the use of a vocabulary anchor (LEL) to build both functional and nonfunctional perspectives. Using this anchor, we first showed how to elicit NFRs building the nonfunctional perspective. We also showed how to integrate NFRs into UML by extending some of the UML sublanguages, and presented a systematic way to integrate NFRs into the functional models.

Other work, like standards [22], [39], [33], offer some guidance on eliciting NFRs. However, these standards basically give different taxonomies for some of the NFRs. The elicitation process per se is shallow. There is also a lack of guidance on how one might integrate the NFRs into design. Similar to our proposal, the Volere Requirements Specification Template [41] brings a deeper comprehension of NFR elicitation. It shows guidelines to elicit NFRs based on questions to be made to each use case found. However, relying only in use cases may facilitate for NFRs to be missed during the elicitation process. Furthermore, deciding on implementing one NFR frequently brings negative impacts to other NFRs. For example implementing a two steps password check can implement the NFR security but will compromise Usability. Since Volere represents NFRs as a statement in the use cases, it does not favor reasoning regarding different alternatives to implement conflicting NFRs (3).

A strong point in our process of integrating NFRs into conceptual models is that we implemented a traceability mechanism. This mechanism provides a way of representing in the models, which aspects are there because of an NFR. This has shown to be quite useful during the model reviewing process. In different situations, the reviewers were surprised by the inclusion of elements in the conceptual models that did not fit their perception of the application; they only became convinced of the necessity by following the traces to the NFR graphs and LEL. The traceability mechanism has also proven to be very helpful during NFRs trade offs since it was easier to check the models to see what possible impacts would arise from dropping one NFR or satisficing another.

This work extends previous works presenting a stronger and more systematic elicitation process. Differently from previous work, this work extends the lexicon to assist NFR elicitation using a lexicon as a natural language-oriented front-end to support the NFR elicitation process. General heuristics for conflict detection are also introduced. The integration process was considerably changed to cope with problems regarding the representation of multiple NFRs for the same class, and also to address dynamic models instead of addressing only static models, as in the previous works. As UML [37] has become a de facto standard for object-oriented modeling, we have used it as the representational schema for conceptual models.

We improved our strategy performing three case studies. The results found in these case studies, together with previous results [14], suggest that the use of this strategy can lead to a final conceptual model with better quality, as well as to a more productive software development process. On the other hand, one of the heuristics used in conflict detection can be very time consuming, impacting the scalability of the approach. The lack of automation between the lexicon use and the construction of the NFR graphs also poses some concerns about the time spent in this task, as well as in the accuracy of the process. Scalability is still an open issue in our strategy. However, we can point out that, as stated in Section 3.3, we have used the strategy in systems where we dealt with more than 70 NFR graphs. As it can be seen from Table 1 (Case Study III), the use of the strategy has not impacted significantly the overall development process and yet stimulated the discovery of several problems in the functional models. Furthermore, applying the strategy to Case Study III (48 classes) had not been more challenging than when we applied it to Case Studies I and II (around 15 classes).

Although our strategy may be used for almost any type of NFR, we understand that its results will be more effective when addressing NFRs that effectively demand actions to be performed by the system, and therefore affects the software design.

NFRs such as Maintenance and Portability are not easily operationalized in an specific point of the artifact, but rather will be more related on how the design is organized. Our

8. The RE-online mailing list aims to act as an electronic forum for exchange of ideas among the requirements engineering researchers and practitioners. http://www-staff.it.uts.edu.au/~didar/RE-online.html.

strategy will help to elicit such NFRs, but since they are not operationalizable, they are not dealt with in our integration proposal.

On the other hand, NFRs such as Safety, Traceability, Performance, Accuracy, and others, frequently demand the design to be carefully studied in order to satisfice these NFRs. Hence, it will be more likely that these NFRs will be the type of NFRs that our strategy will help the most. It is important to remember that we do not propose to factor out the NFR once they are integrated in the functional model. This issue of factoring out NFRs is being treated by the aspect literature [24]. We envision future work as dealing with other UML artifacts and performing new case studies in an independent manner, which is without the participation of any of us, to verify how easily this strategy can be applied by other developers. In terms of long-range research we hope that others, and ourselves, focus on how to automate some of the processes we described here. Of course, there are several challenges, but approaches that could handle the nonfunctional requirements knowledge base as a basis for an intelligent assistant, as well as to help identify possible conflicts areas, certainly would contribute to the adoption of NFR elicitation processes by software organizations.

Acknowledgments

This work was partially supported by NSERC grant and CNPq and by FAPERJ: Cientista do Nosso Estado grants. This work has been partially introduced in [12]. Dr. Leite also acknowledges the support of the University of Toronto and of CAPES since part of the final editing of the paper was done during a sabbatical leave at University of Toronto.

References

[1] V.R. Basili, R.W. Selby, and D.H. Hutchens, "Experimentation in Software Engineering," *IEEE Trans. Software Eng.*, vol. 12, no. 7, pp. 733-742, July 1986.

[2] B. Boehm, *Characteristics of Software Quality.* North Holland Press, 1978.

[3] B. Boehm and H. Hoh, "Identifying Quality-Requirement Conflicts," *IEEE Software*, pp. 25-36, Mar. 1996.

[4] E. Börger and R. Gotzhein, "Requirements Engineering Case Study 'Light Control,'" http://rn.informatik.uni-kl.de/recs, 2002.

[5] K.K. Breitman, J.C.S.P. Leite, and A. Finkelstein, "The World's Stage: A Survey on Requirements Engineering Using a Real-Life Case Study," *J. the Brazilian Computer Soc.*, vol. 6, no. 1, pp. 13-38, July 1999.

[6] K.K. Breitman, "Evolução de Cenários," PhD thesis, Pontificia Univiversidade Católica do Rio de Janeiro, May 2000.

[7] F.P. Brooks Jr., "No Silver Bullet: Essences and Accidents of Software Engineering," *Computer*, no. 4, pp. 10-19, Apr. 1987.

[8] L. Chung, "Representing and Using Nonfunctional Requirements: A Process Oriented Approach," PhD thesis, Dept. of Computer Science, Univ. of Toronto, June 1993, also Technical Report DKBS-TR-91-1.

[9] L. Chung and B. Nixon, "Dealing with Nonfunctional Requirements: Three Experimental Studies of a Process-Oriented Approach," *Proc. 17th Int'l Conf. Software Eng.*, pp. 24-28, Apr. 1995.

[10] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering.* Kluwer Academic, 2000.

[11] L.M. Cysneiros and J.C.S.P. Leite, "Integrating Non-Functional Requirements into Data Model," *Proc. Fourth Int'l Symp. Requirements Eng.*, June 1999.

[12] L.M. Cysneiros and J.C.S.P. Leite, "Using UML to Reflect Nonfunctional Requirements," *Proc. 11th CASCON Conf.*, pp. 202-216, Nov. 2001.

[13] L.M. Cysneiros and E. Yu, "Non-Functional Requirements Elicitation," *Perspective in Software Requirements,* Kluwer Academics, 2003.

[14] L.M. Cysneiros, J.C.S.P. Leite, and J.S.M. Neto, "A Framework for Integrating Nonfunctional Requirements into Conceptual Models," *Requirements Eng. J.*, vol. 6, no. 2, pp. 97-115, 2001.

[15] A. Davis, *Software Requirements: Objects Functions and States.* Prentice Hall, 1993.

[16] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach,* second ed. Int'l Thomson Computer Press, 1997.

[17] A. Finkelstein and J. Dowell, "A Comedy of Errors: The London Ambulance Service Case Study," *Proc. Eighth Int'l Workshop Software Specification and Design*, pp. 2-5, 1996.

[18] M. Fowler and K. Scott, *UML Distilled.* Addison-Wesley, 1997.

[19] J. Goguem and C. Linde, "Techniques for Requirements Elicitation," *Proc. First Int'l Symp. Requirements Eng.*, pp. 152-164, 1993.

[20] D.J. Grimshaw, W. Godfrey, and G.W. Draper, "Non-Functional Requirements Analysis: Deficiencies in Structured Methods," *Information & Software Technology*, vol. 43, no. 11, pp. 629-635, 2001.

[21] G. Hadad et al., "Construcción de Escenarios a partir del Léxico Extendido del Lenguage," *Proc. Jornadas Argentinas de Informática e Investigacón Operativa (JAIIO'97)*, pp. 65-77, 1997.

[22] IEEE Recommended Practice for Software Requirements Specification, Standard for Information Technology IEEE, 1998.

[23] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process.* Addison-Wesley Longman, 1999.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming (ECOOP'97)*, June 1997.

[25] T.G. Kirner and A.M. Davis, "Nonfunctional Requirements of Real-Time Systems," *Advances in Computers*, vol. 42, pp. 1-38, 1996.

[26] D.R. Lindstrom, "Five Ways to Destroy a Development Project," *IEEE Software*, pp. 55-58, Sept. 1993.

[27] J.C.S.P. Leite and A.P.M. Franco, "A Strategy for Conceptual Model Acquisition," *Proc. First IEEE Int'l Symp. Requirements Eng.*, pp. 243-246, 1993.

[28] J.C.S.P. Leite et al., "Enhancing a Requirements Baseline with Scenarios," *Requirements Eng. J.*, vol. 2, no. 4, pp. 184-198, 1997.

[29] C. Leonardi et al., "Una Estrategia de Análisis Orientada a Objetos Basada en Escenarios," *Proc. Actas II Jornadas de Ingeniaria de Software (JIS97)*, Sept. 1997.

[30] *Handbook of Software Reliability Engineering.* M.R. Lyu, ed., McGraw-Hill, 1996.

[31] J. Musa, A. Lannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, 1987.

[32] J. Mylopoulos, L. Chung, E. Yu, and B. Nixon, "Representing and Using Non-Functional Requirements: A Process-Oriented Approach," *IEEE Trans. Software Eng*, vol. 18, no. 6, pp. 483-497, June 1992.

[33] NASA Software Document Standard (NASA-STD-2100-91), 1991.

[34] J.S.M. Neto, "Integrando Requisitos Não Funcionais ao Modelo de Objetos," MSc dissertation, Pontificia Univ. Católica do Rio de Janeiro, Mar. 2000.

[35] Rational, "Object Constraint Language Specification," 1997, http://www.rational.com.

[36] D. Ross, "Structures Analysis: A language for Communicating Ideas," *IEEE Trans. Software Eng*, vol. 3, no. 1, pp. 16-34, Jan. 1977.

[37] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[38] H.A. Simon, *The Sciences of the Artificial*, third ed. MIT Press, 1996.

[39] US Department of Defense System Software Development, DOD Standard 2167A, 1997.

[40] A. Van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour," *Proc. Fifth Int'l Symp. Requirements Eng.*, pp. 249-262, 2001.

[41] S. Robertson and J. Robertson, *Mastering the Requirements Process.* ACM Press, 1999.

[42] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software.* Prentice Hall, 1990.

Luiz Marcio Cysneiros received the PhD degree from the Pontificia Universidade Católica do Rio de Janeiro, Brazil, in 2001 and has been involved with requirements engineering since 1996. He is an assistant professor at York University, Toronto. He has held a postdoc position at the University of Toronto from April 2001 to July 2002, where he continued his studies on nonfunctional requirements. He has published papers at several requirements related conferences, including RE'99, and in the *Requirements Engineering Journal*. He also has been presenting tutorial on nonfunctional requirements in many international conferences such as ICSE '02, RE '03, and UML '03. He has an extensive industrial experience, mainly in a computer manufacturer and on health care domain. He is a member of the IEEE Computer Society.

Julio Cesar Sampaio do Prado Leite received the PhD degree at the University of California, Irvine, in 1998. He is an associate professor at Pontificia Universidade Católica do Rio de Janeiro, Brazil, and has been involved in requirements engineering, specifically, since his pioneer work on viewpoints. He has been program chair and conference chair for several conferences and has more than 90 full papers published at conferences proceedings and 19 journal papers. He is a member of the IFIP W.G. 2.9 on software requirements engineering, member of the IEEE committee on software reuse, and is an associate editor for the *Requirements Engineering Journal*. He is a founding member of the Brazilian Computer Society, member of the IEEE Computer Society, and of the ACM.

▹ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.