An Approach for Reasoning and Refining Non-Functional Requirements

Nelson Souto Rosa, Paulo Roberto Freire Cunha
Centro de Informática
Universidade Federal de Pernambuco
Caixa Postal 7851 Cidade Universitária
50740-540 Recife, Pernambuco - Brazil
Phone: +55 81 3271 8430 Fax: +55 81 3271 8438
E-mail: {nsr,prfc}@cin.ufpe.br

George Roger Ribeiro Justo
Cap Gemini Ernst & Young UK
George.Justo@capgemini.co.uk

Abstract

Non-functional requirements (NFRs) are rarely taken in account in most software development processes. There are some reasons that can help us to understand why these requirements are not explicitly dealt with: their complexity, NFRs are usually stated only informally, their high abstraction level and the rare support of languages, methodologies and tools. In this paper, we concentrate on defining how to reason and how to refine NFRs during the software development. Our approach is based on software architecture principles that guide the definition of the proposed refinement rules. In order to illustrate our approach, we adopt it to an appointmernt system.

Keywords: Software Architecture, Non-Functional Requirements, Refinement Rules.

1 Introduction

Functional requirements define *what* a software is expected to do. Non-functional requirements (NFRs[1] define *how* the software operates or how the functionality is exhibted [4]. Functional requirements typically have localised effects, i.e., they af ect only the part of the software addressing the functionality defined by the requirement. On the other h nd, NFRs typically specify global constraints that must be satisfied by the software, e.g., *performace*, *fault-tolerance*, *availability*, *security* and so on. During the software development process, functional requirements are usually incorporated into the software artefacts step by step. At the end of the process, all functional requirements must have been implemented in such way that the software completely satisfies the requirements defined at the early stages. NFRs, however, are not implemented in he same way as the functional ones. They are usually satisfied in a certain degree, or *satisficed* [28], as a consequence of design decisions taken to implement the software's functionality.

NFRs are rarely considered when a software is built, especially in the early stages of the software development process. There are some reasons that can help us to understand why these requirements are not explicitly dealt with: NFRs are usually very abstract and stated only informally, e.g., "the system must have a satisfactory *performance*" or "the component is *secure*" are common descriptions of NFRs; NFR are rarely supported by languages, methodologies and tools; NFRs are more complex to deal with; NFRs are difficult to be effectively carried out during the software development; it is not trivial to verify wheter a specific NFR is satisfied by the final product or not, i.e., it is difficult to validate them in the final product; very often NFRs conflict and compete with each other, e.g., *availability* and *peformance*; NFRs commonly concern environment builders instead of application programmers; and the separation of functional and non-functional requirements is not easily defined.

In spite of these difficulties, the necessity of dealing explicitly with NFRs is apparent [28, 8, 37]. Firstly, there is an increasing demand for fault-tolerant, multimedia and real-time applications, in which NFRs play a critical

[1] Also referred to as goals [28], afunctional qualities[9] , *ilities* [11], softgoals [8] or software quality factors [15]).

role and their satisfaction is mandatory. Secondly, as a kind of requirement, it is natural their integration into the software development process. Thirdly, interactions among functional and non-functional requirements are so strong in most cases that NFRs can not be satisfied just as a consequence of design decisions taken to satisfy the functional requirements. Finally, an explicit treatment of NFRs enables us to predict some quality properties of the final product in a more reasonable and reliable way [37].

In order to address the problem of explicitly dealing with NFRs, two approaches have been traditionally adopted: process-and product-oriented [28]. In the first approach, NFRs are viewed as effective elements in the software development process as they are considered together with functional requirements to guide the construction of the software. In the product-oriented approach, NFRs are determined in the final product where they are explicitly stated. In this approach, NFRs are measured and used to compare quality attributes of the software.

On this scenario, we present a process-oriented approach in which functional and non-functional requirements are refined together in a semi-formal way. In order to carry out this task, our approach has been defined around basic abstractions that represent our comprehension of NFRs and implementation elements, together with their relationships. Additionally, we adopt software architecture principles as the basis of our proposal. In fact, the software architecture serves as the integration point of functional and non-functional requirements. It is worth observing that our approach concentrates on structural aspects of software architecture, instead their behaviour.

This paper is organised as following: Section 2 presents our view on how to reason about NFRs. It includes a set of abstractions and the notion of conflict and correlation between NFRs. Section 3 presents the proposed refinement rules. Next, in order to illustrate our approach, we adopt it to an appointment system that must be designed over an object-oriented middleware [16]. Section 5 presents some related works. Finally, the last section presents the conclusions and some directions for future work.

2 Formalising Basic Concepts

Our approach explicitly addresses non-functional requirements throughout the development of software systems. It defines how non-functional properties are expressed, integrated into architecture-based development and refined.

In order to address these tasks, we concentrate on the following elements:

- **Abstractions**: a set of abstractions (NF-Abstractions), namely NF-Attributes, NF-Statements, and NF-Actions, is provided and systematically defined in order to reason and model non-functional properties [36];
- **Integration**: NF-Abstractions are integrated into software architectures (elements of the software architecture model functional elements of systems). At the end of the integration, an abstract non-functional software architecture (NF-Architecture) is defined containing non-functional properties associated with architectural elements [36]; and
- **Refinement**: the abstract architecture is refined according to the proposed refinement rules. Structural and non-functional characteristics of dynamic distributed systems are refined into more concrete elements closer to actual implementation. From the abstract non-functional software architecture, the refinement yields a concrete non-functional software architecture that is realisable [37];

There are some characteristics that represent the core of our approach. Firstly, the set of abstractions is the basis of our approach. Secondly, software architecture is used to represent the structure of the system, while its abstractions serve as integration points for the non-functional properties. Thirdly, the abstract non-functional software architecture is the basis of the design steps that follow. Hence, its refinement means that structural (software architecture) and non-functional issues are taken into account together. Fourthly, the inclusion of non-functional properties brings additional due when the software architecture has to be actually implemented.

Next, we present the proposed abstractions for modelling NFRs.

2.1 Non-Functional Attribute

Non-functional attributes (NF-Attributes) model non-functional characteristics that can be precisely measured such as *performance*; non-functional features that cannot be quantified, but may be defined as required in the final product to a certain level, such as *security* and *availability;* and any non-functional aspect that must simply be present (without measure or level), like the transaction properties *atomicity*, *consistency*, *isolation* and *durability.*

Another key characteristic of NF-Attributes is the possibility of decomposing them. An NF-Attribute is usually decomposed into primitive NF-Attributes that are more detailed or closer to implementation elements. The decomposition serves to differentiate NF-Attributes referred to **simple** or **composite**. A simple NF-Attribute is not decomposed, while a composite one is broken up into more primitive NF-Attributes. For a composite NF-Attribute, its primitive components participate in three different ways in order to make up the NF-Attribute:

- all primitive attributes are necessary in the definition of the NF-Attribute;
- at least one (any) primitive attribute is necessary in the definition of the NF-Attribute; and
- exactly one primitive attribute is necessary in the definition of the NF-Attribute.

The degree of decomposition of an NF-Attribute usually depends on both the acquired knowledge about the application domain and the NF-Attribute itself. For example, a real-time system is expected to demand *performance* characteristics in a more detailed manner than most common applications, as it is a key non-functional property to be considered. Another example is a safety-critical application, in which *security* aspects must be extensively known and decomposed in order to be effectively achieved in the final product.

Finally, as there is a great number of NF-Attributes and their variety is enormous, we have decided to focus on those related to runtime issues. For example, *performance*, *availability* and *security* are perceived when the software is already running, rather than issues of its development (non-runtime quality attributes [9]) such as *reusability*, *testability* and *modifiability*. The decision to select runtime properties has been motivated by certain facts: most of them are included in the software requirement specification [17]; they are more visible to user application than that of developers (e.g., the *modularity* typically concerns only application developers, whilst the *usability* is essential to users); their satisfaction has been increasing in most WEB applications such as the Internet search engines; they are critical factors for the proper functioning of real-time and safety-critical systems; and they are directly affected by functional requirements.

NF-Attributes are defined as follows.

Definition 2.1 (NF-Attribute) *An NF-Attribute is a tuple* $att = (S, Cont)$*, where*

- S *is a subset of the primitive runtime attributes* ($S \subseteq RT\_Attributes$) *that composes* att*, where*
  - $RT\_Attributes$ *is a subset of attributes related to execution time* ($RT\_Attributes \subset Attributes$)*, where*
    - $RT\_Attributes$ = *{accessibility, accountability, accuracy, adaptability, auditability, availability, confidentiality, configurability, consistency, controllability, efficiency, fault-tolerance, flexibility, generability, integrity, maturity, mobility, performance, reconfigurability, recoverability, reliability, repleaceability, robustness, safety, security, space-performance, throughput, time_performance, tolerance, traceability, transparency}* [8],
- $Cont$ ($Cont \in \Delta_{Cont}$) *defines the kind of contribution of primitive attributes to* att
  - $\Delta_{Cont} = \{all, one^{+}, one^{X}, none\}$*, where*
    - all*: att is completely defined by composing all its primitive attributes,*
    - one^{+}*: att is completely defined by one or more of its primitive attributes,*
    - one^{X}*: att is completely defined by exactly one (any) of its primitive attributes,*
    - $none$*: att has no primitive attributes, i.e., it is already completely defined.*

Two operators are defined in order to access the elements of the NF-Attribute:

- attribute.att yields S,
- compositionOfAttributes.att yields $Cont$.

Three basic rules are derived from the previous definition:

- **R1** $att = (S, Cont)$, $Cont \neq none \Leftrightarrow S \neq \emptyset$, att is a composite NF-Attribute,
- **R2** $att = (S, Cont)$, $Cont = none \Leftrightarrow S = \emptyset$, att is a simple NF-Attribute,
- **R3** Two composite NF-Attributes att_1, $att_2 \in RT\_Attributes$, $att_1 \neq att_2$, have no primitive NF-Attributes in common: attributes.$att_1 \cap$ attributes.$att_2 = \emptyset$.

Example 2.1 (Performance) *The NF-Attribute performance is* (S, $Cont$)*, where*

- S = *{time_performance,space_performance}*,
- $Cont$ = all,
- attributes.$performance = \{time\_performance, space\_performance\}$,
- compositionOfAttributes.$performance = all$.

$Performance$ is typically decomposed into more primitive NF-Attributes with respect to processing time ($time\_performance$) and storage needs ($space\_performance$).

2.2 Non-Functional Action

A Non-Functional Action (NF-Action) models either any software aspect or any hardware characteristics that affect the NF-Attribute. Software aspects mean design decisions, algorithms, data structures and so on, while hardware features concern computer resources available for running the software system. For instance, the NF-Attribute *performance* is decomposed into two additional more primitive NF-Attributes, namely *space_performance* and *time_performance*. An algorithm used to compress data has a direct influence on the primitive NF-Attribute *space_performance*, meanwhile hardware characteristics such as size of the main memory and the secondary storage capacity also affect the *space_performance*.

An important issue to be considered in the previous definition of the NF-Action is the meaning of the statement "an NF-Action affects an NF-Attribute ". A more precise consideration of "affects" reveals that it refers to having an "effect on" or "realise" a non-functional aspect. By realising, the NF-Action acts in order to operationalise what is defined in the NF-Attribute. In relation to "effect", it refers to having an influence, i.e., an action whose effect on the NF-Attribute cannot be neglected. The effect on an NF-Attribute is either *against* or *in favour* of it. For instance, a *good performance* is not implemented directly, but there are NF-Actions that affect the *performance* and may be implemented in order to achieve it. Unlike *performance*, *security* is not simply affected by encryption algorithms or authorisation access, but it is actually implemented by them.

Another basic issue related to NF-Actions is the notion of "correlation". Correlation refers to the fact that an NF-Action implementing or affecting a particular NF-Attribute may also have (or may be correlated with) an effect on other NF-Attributes. Again, the NF-Action may act *against* or *in favour* of other NF-Attributes. For instance, NF-Action *encryption algorithm* implements aspects of the NF-Attribute *security*. However, this NF-Action also interferes in the NF-Attribute *performance*, as it is necessary to spend time to execute the encryption.

As mentioned before, there are very often conflicts between NFRs. These conflicts appear as the result of correlations between NF-Attributes. For instance, if an NF-Action is essential for implementing *good performance*, but has a strong negative effect on another NF-Attribute, may it be implemented or not? We deal with this kind of situation by allowing the definition of priorities in how the constraints imposed on NF-Attributes must be considered. Thus, whatever the impact of an NF-Action on an NF-Attribute, it must be implemented according to the priority assigned to the NF-Attribute.

NF-Actions are defined as follows:

Definition 2.2 (NF-Action) *An NF-Action is a tuple* $act = (A, I, e)$, *where*

- A *is the set of NF-Attributes affected by act,*
- I *is the set of NF-Attributes that are directly realised by act,*
- e, *a function that defines the kind of effect of act on the NF-Attributes, is defined as* $e : A \rightarrow \Delta_{Effect} \times \Delta_{Degree}$, *where (given* $att \in A$*)*
  - $\Delta_{Effect} = \{against, inFavour\}$, *where*
    - *against: the implementation of the act causes a negative impact on att,*
    - *inFavour: the implementation of the act causes a positive impact on att.*
  - $\Delta_{Degree} = \{+3, +2, +1, -1, -2, -3\}$, *where*
    - +3 *a high inFavour effect on att,*
    - +2 *a medium inFavour effect on att,*
    - +1 *a low inFavour effect on att,*
    - -1 *a low against effect on att,*
    - -2 *a medium against effect on att,*
    - -3 *a high against effect on att.*

It is worth pointing out that the levels of effect introduced in the previous definition have been proposed by adopting a basic criteria: an step necessary to effectively treat NFRs consists of defining mechanisms that can be used to precisely quantify (as much as possible) them. Hence, it is necessary to define degrees of effect of NF-Actions over the NF-Attributes, instead of simply to identify *in favour* and *against* effects, i.e., it is also necessary to quantify this effect.

Additionally, some auxiliary elements are defined in order to access the elements of the NF-Action:

- affectedAttributes.*act* yields A,

- implementedAttributes.act yields I,
- kindOfEffect.$act(att)$ yields $first(e(att))$, where $first(RXT)$ returns $r \in R$,
- degreeOfEffect.$act(att)$ yields $second(e(att))$, where $second(RXT)$ returns $t \in T$,
- Let $att \in Attributes$, $IsAffected(att)$ returns true if $att \in A$,
- Let $att \in Attributes$, $IsImplemented(att)$ returns true if $att \in I$.

Five basic rules are derived from the previous definition:

- **R1** $att \in A$, kindOfEffect.$act(att) = against \Rightarrow$ degreeOfEffect.$act(att) \in$ {-1,-2,-3},
- **R2** $att \in A$, kindOfEffect.$act(att) = inFavour \Rightarrow$ degreeOfEffect.$act(att) \in$ {+1,+2,+3},
- **R3** An NF-Action either implements or affects an NF-Attribute att: $att \in$ implementedAttributes.$act \Rightarrow att \notin$ affectedAttributes.act or $att \in$ affectedAttributes.$act \Rightarrow att \notin$ implementedAttributes.act,
- **R4** An NF-Action $acts$ either implements or affects at least one NF-Attribute: $\exists\ att \in RT\_Attribute\ att \in$ affectedAttributes.act or $att \in$ implementedAttributes.act,
- **R5** An NF-Action act affects or implements simple NF-Attributes: $\forall\ att \in RT\_Attributes$, $att \in$ {implementedAttributes.$act \cup$ affectedAttributes.act} $\Rightarrow$ attributes.$att = \emptyset$.

Example 2.2 *The NF-Action useIndexing* [8] is an act, where

- A = $\{time\_performance\}$,
- I = {},
- e = $\{(time\_performance, inFavour, +3)\}$,
- affectedAttributes.$act = \{time\_performance\}$,
- implementedAttributes.act = {},
- kindOfEffect.$act(time\_performance) = inFavour$,
- degreeOfEffect.$act(time\_performance)$ =+3,
- $IsImplemented(time\_performance) = false$,
- $IsAffected(time\_performance) = true$.

In this example, the implementation of *useIndexing* acts in order to improve $time\_performance$ and its effect is high (+3). It is worth observing that *useIndexing* does not implement $time\_performance$, but only affects it.

Example 2.3 *The NF-Action AuthoriseAccess is (A, I, e), where*

- $A = \{time\_performance\}$,
- $I = \{security\}$,
- $e = \{(time\_performance, against, -1)\}$.

In this example, *AuthoriseAccess* is an effective implementation of a *security* mechanism, while it also affects the *time_performance*. Observing examples 2.2 and 2.3, it is important to note the difference between an NF-Action that acts in order to implement an NF-Attribute and an NF-Action that affects an NF-Attribute. Unlike NF-Actions related to *security*, ones associated with *performance* only act $inFavour$ or $against$ it. This is often the kind of situation in which non-functional properties are considered.

Definition 2.3 (Correlation) *Two NF-Attributes att_1, att_2 are correlated if given an NF-Action act then att_1, $att_2 \in$(*affectedAttributes.$act \cup$ implementedAttributes.act*).*

Example 2.4 *A well-known example of correlated NF-Attributes is performance and security. An NF-Action that implements an authorisation access necessary to achieve a certain level of security acts against performance, i.e., the time spent in the authorisation affects performance.*

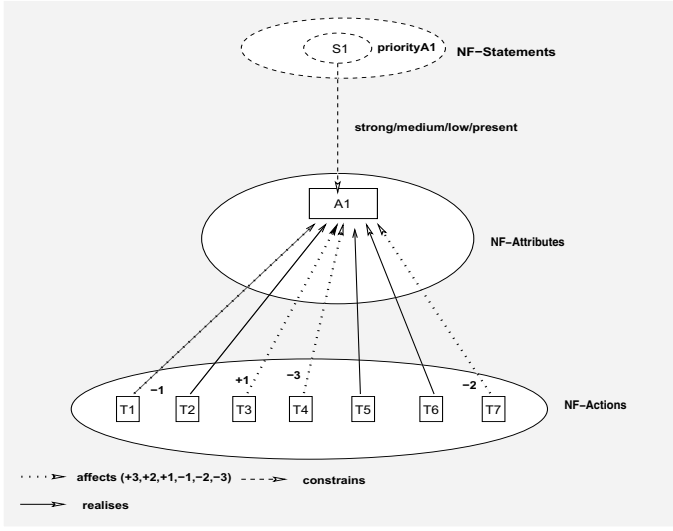

Figure 1: Constraints imposed by NF-Statements and their consequence on the selection of NF-Actions

2.3 Non-Functional Statement

A Non-functional statement (NF-Statement) represents constraints on possible design decisions taken to implement a software system. It means that every design decision made in order to implement any functionality of the software must respect the constraints imposed by the non-functional part of the requirements.

NF-Statements represent those constraints over the NF-Attributes, e.g., *good performance* is a constraint on the NF-Attribute *performance*. In practical terms, the constraint imposed by the NF-Statement defines a subset of NF-Actions that can be used to satisfy it. For example, any NF-Action that affects the NF-Attribute *performance* may be implemented or not depending on the NF-Statement.

As mentioned before, our approach adopts the notion of satisfiability proposed in [28], in which NF-Statements are achieved to a certain degree, rather than absolutely. Essentially, given a set of NF-Actions that affect/implement an NF-Attribute, the satisfaction of an NF-Statement consist of implementing the NF-Actions, according to the following kind of constraints:

- *strong*: the *strong* constraint defines that the satisfaction of an NF-Statement requires to every NF-Action that realises the NF-Attribute must be considered, every NF-Action that has an *inFavour* effect (+3,+2,+1) over the NF-Attribute must be considered and no NF-Action that has an *against effect* (-1,-2,-3) over the NF-Attribute is considered.

- *medium*: the *medium* constraint defines that the satisfaction of an NF-Statement requires to every NF-Action that realises the NF-Attribute must be considered, every NF-Action that has an *inFavour* effect (+3,+2,+1) over the NF-Attribute must be considered and every NF-Action that has an *against* effect of kind -1 and -2 over the NF-Attribute must be considered,

- *weak*: the *weak* constraint defines that the satisfaction of an NF-Statement requires to every NF-Action that realises the NF-Attribute must be considered, every NF-Action that has an *inFavour* (+1,+2,+3) effect must be considered and every NF-Action that has an *against* effect (-1,-2,-3) must be considered,

- *present*: the *present* constraint defines that the satisfaction of an NF-Statement requires to only NF-Actions that realise the NF-Attribute must be considered.

For instance, the NF-Statement *secure* may be defined as a *strong* constraint over the NF-Attribute *security*, while the NF-Attribute *atomicity* may be defined as *present* in the NF-Statement *ACID*. Another main issue to be considered refers to NF-Statements that constrain more than one NF-Attribute. It is necessary to define priorities that can be used to solve conflicts caused by correlated NF-Attributes.

Figure 1 shows an example in which a NF-Statement (S1) defines which NF-Actions must be considered according to the constraint imposed over the NF-Attribute A1:

- a *strong* constraint over A1 means that the NF-Actions T2, T3, T5 and T6 must be realised;

- a *medium* constraint over A1 means that the NF-Actions T1, T2, T3, T5, T6 and T7 must be realised;

- a *weak* constraint over A1 means that the NF-Actions T1, T2, T3, T4, T5, T6 and T7 must be realised;
- a *present* constraint over A1 means that the NF-Actions T2, T5 and T6 must be realised.

An NF-Statement is defined as follows:

Definition 2.4 (NF-Statement) *An NF-Statement is a tuple sta = (S, c, p), where*

- *S (S ⊆ RT_Attributes) is the set of NF-Attributes constrained by sta,*
- *c, a function that specifies the kind of constraint imposed on each NF-Attribute, is defined as c : S →$\Delta_{Constraint}$*
  - *$\Delta_{Constraint}$ = {weak,medium,strong,present},*
- *p, a function that defines the priority of each NF-Attribute constrained in the NF-Statement, is defined as p : S → $\Delta_{Priority}$*
  - *$\Delta_{Priority}$ = {low,medium,high}.*

Some operators have been defined in order to access the elements of the NF-Statement:

- constrainedAttributes.*sta* returns S,
- kindOfConstraint.*sta*(*att*) returns the kind of constraint over *att*,
- priority.*sta*(*att*) returns the priority of *att*.

Three basic rule has been derived from the previous definition of the NF-Statement:

- **R1** An NF-Statement *sta* constrains at least one NF-Attribute: ∃ *att* ∈ *RT_Attributes* | *att* ∈ constrainedAttributes.*sta*
- **R2** Given the NF-Statement *sta* = (S, c, p), an NF-Action *act* and *att* ∈ constrainedAttributes.*sta*, the NF-Actions actually taking into account (*Actions*) in order to achieve the constraint imposed by *sta* are defined as follows:
  1. *c(att)* = *strong*
     - *Implemented_Actions* = {∀ *act* ∈ *Actions* | *att* ∈ implementedAttributes.*act* ∨ (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = +3) ∨ (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = +2) ∨ (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = +1)}.
  2. *c(att)* = *medium*
     - *Implemented_Actions* = {∀ *act* ∈ *Actions* | *att* ∈ implementedAttributes.*act* ∨ (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = +3) ∨ (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = +2) ∨ (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = +1)} ∪ {∃ *act* ∈ *Actions* | (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = -1) ∨ (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = -2) }.
  3. *c(att)* = *weak*
     (a) *Implemented_Actions* = {∀ *act* ∈ *Actions* | *att* ∈ implementedAttributes.*act*} ∨ {∃ *act* ∈ *Actions* | (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = +3) ∨ (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = +2) ∨ (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = +1)} ∪ {∃ *act* ∈ *Actions* | (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = -1) ∨ (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = -2) ∨ (*att* ∈ affectedAttribute.*act* ∧ kindOfEffect.*act*(*att*) = -3)}.
  4. *c(att)* = *present*
     (a) *Actions* = {∀ *act* ∈*Actions* | *att* ∈ implementedAttributes.*act*}.

R3 (Conflict) Given the NF-Action *act*, NF-Statement *sta* and att_1, att_2 ∈ constrainedAttributes.*sta*,

- *act* ∈ *Implemented_Actions*,
- kindOfEffect.*act*(att_1) ≠ kindOfEffect.*act*(att_2).

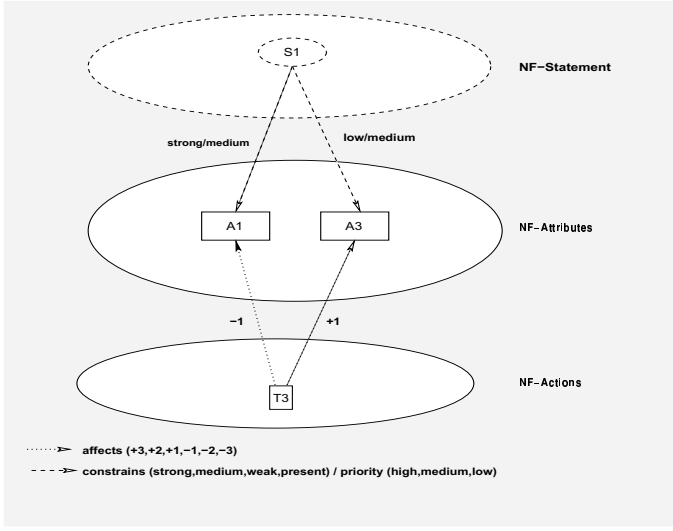

Figure 2: Conflict between NF-Actions

Figure 2 shows a situation of conflict between the constraints imposed over A1 and A2. In this case, the constraint imposed over A1 (*strong*) defines that T1 cannot be implemented, whilst the constraint imposed over A2 (*low*) defines that T1 must be implemented. The priority assigned to each NF-Attribute may solve the conflict, e.g., if the priority assigned to A1 is higher than one associated to A2 then T1 is not implemented, otherwise T1 is implemented.

Example 2.5 *The NF-Statement Fast_High_Secure is* (S, c, p)*, where*

- S = *{performance,security},*
- c = *{(performance, medium), (security,high)},*
- p = *{(performance, low), (security,high)}.*

In this example, every NF-Action that realises the NF-Attribute *security* must be considered, every NF-Action that affects positively the NF-Attribute *security* must be considered and no NF-Action that affects negatively (-1,-2,-3) the NF-Attribute *security* must be considered (*high* constraint imposed over the NF-Attribute *security*). On the other hand, every NF-Action that realises the NF-Attribute *performance* must be considered, every NF-Action that affects positively (+1,+2,+3) the NF-Attribute *performance* must be considered and every NF-Action that has an *against* effect of kind -1 and -2 over the NF-Attribute *performance* must be considered (*medium* constraint imposed over the NF-Attribute *performance*). Additionally, the priority assigned to *security* is higher than one defined to *performance*, which means that the NF-Actions related to (affect/implement) *security* must be considered prior to ones related to *performance*.

Figure 3 shows the relationship between the abstractions presented in the previous sections. According to this figure, it is possible to observe that:

- the NF-Attributes A11 and A12 are primitive NF-Attributes of the NF-Attribute A1,
- A12 and A221 are correlated,
- there is a conflict in the implementation of T3,
- the NF-Actions T1 and T2 realise the NF-Attribute A11,
- the NF-Action T3 affects the NF-Attributes A12 and A221,
- the NF-Actions (T1,T2,T4,T5) only implement and affect simple NF-Attributes (A11,A221),
- the NF-Statement S2 constrains the NF-Attributes A1 and A2.

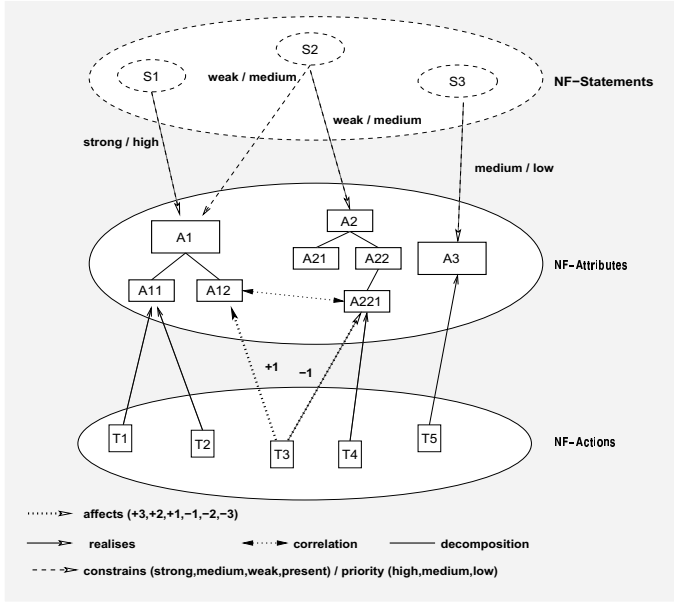

Figure 3: Relationships Among Non-Functional Abstractions

2.4 Integration with Software Architecture

In this part of our approach a strategy on how to integrate functional and non-functional elements is defined. It defines that NF-Statements (defined in Section2.3) are assigned to architectural elements, namely connectors, components, ports and the entire configuration. In terms of development, it means that the functional part comes to be constrained by the non-functional one. From this point, any development of components and connectors must respect the constraints imposed by the NF-Statement. We name non-functional software architecture (NF-Architecture), a software architecture in which non-functional properties are explicitly defined. Similarly, we designate NF-Port, NF-Interface NF-Component and NF-Connector to refer to Port, Interface, Component and Connector that have non-functional properties assigned to them, respectively.

Prior to present the integration, we firstly define the architectural elements mentioned above.

Definition 2.5 (Port) *A port is a tuple $p = (type)$, where*

- *$type \in$ {entry,exit} defines the kind of the port.*

Definition 2.6 (Interface) *An Interface is a set of ports $int = \{p_1, ..., p_n\}$, where*

- *p_i is a port.*

Definition 2.7 (Component) *A component is a tuple $comp = (I_c)$, where*

- *I_c is an Interface.*

An operator is defined to access the elements of the component as follows:

- ports.*comp* yields the set of ports of the component *comp*.

Definition 2.8 (Connector) *A connector is a tuple $conn = (I_t)$, where*

- *I_t is an Interface.*

An operator is defined to access the elements of the connector as follows:

- ports.*conn* returns the set of ports of the connector *conn*.

Definition 2.9 (Architecture) *An architecture is a tuple $arc = (Cmp, Cnt, Cnc)$, where*

- *$Cmp = \{c_1, ..., c_n\}$ is the set of components that make up the architecture,*

- $Cnt = \{t_1, ..., t_n\}$ *is the set of connectors that make up the architecture,*
- $Cnc = \{(c_i, t_k, c_j), ..., (c_r, t_m, c_s)\}$ *is the set of connections of the software architecture.*

Additionally, some architectural operators are defined as follows:

- components.arc returns the set of components that make up the software architecture arc,
- connectors.arc returns the set of connectors that composes the architecture arc,
- connections.arc returns the connections of the architecture arc.

Definition 2.10 (NF-Element) *A $NF\_Element = Element \succ sta$ is a non-functional architectural element, where*

- *Element ∈ {p, int, comp, conn, arc}, where p, int, comp, conn and arc are the architectural elements port, interface, component, connector and architecture, respectively.*

The semantics of the "integration" operator $\succ$ is informally stated as "*Element* is implemented respecting the constraints imposed by the NF-Statement". The NF-Architecture is the coarse-grain non-functional element, while the port represents the fine-grain one. In this way, the basic idea is that constraints imposed on the *Architecture* are applied to their *Components* and *Connectors*. Similarly, constraints imposed on *Components* and *Connectors* are applied to their respective *Ports*, and constraints imposed on the interface are also applied to their *Ports*.

Formally, the integration of architectural and non-functional properties is defined through the following rules:

R1 (Interface integration) Let $Int = \{p_1, ..., p_n\}$ and the NF-Statement sta,
$nf\_int = int \succ sta$ defines that
$int \succ sta \Rightarrow p_1 \succ sta \wedge ... \wedge p_n \succ sta$.

This rule defines that the constraint imposed on the interface is also applied to each port that make up the interface.

R2 (Component integration) Let $comp = (I_c)$ and the NF-Statement sta,
$nf\_comp = comp \succ sta$ defines that
$comp \succ sta \Rightarrow I_c \succ sta$.

This rule defines that the constraint imposed on the component is applied to its interface.

R3 (Connector integration) Let $conn = (I_t)$ and the NF-Statement sta,
$nf\_conn = conn \succ sta$ defines that
$conn \succ sta \Rightarrow I_t \succ sta$.

In a similar way as the previous rule, the constraint imposed on the connector is also applied to its interface.

R4 (Architecture integration) Let $arc = (\{c_1,...,c_n\},\{t_1,...,t_m\},\{(c_i,t_j,c_{i+1}),...,(c_{i+2},t_{j+1},c_{i+3})\}$ and the NF-Statement sta,
$nf\_arc = arc \succ sta$ defines that
$arc \succ sta \Rightarrow c_1 \succ sta \wedge c_1 \; ... \; c_n \succ sta \wedge t_1 \succ sta_1 \wedge ... \wedge t_m \succ sta$.

This rule defines that the constraint imposed on the software architecture is applied to all its components and connectors.

In the integration process, it is worth observing that: NFRs are usually assigned to components and connectors, rather than ports, interfaces and configuration; and the integration must occur before any refinement of the functional part, as the constraint imposed by the NFRs can affect the design of architectural elements.

3 Refining Non-Functional Software Architectures

After being described and integrated with software architecture elements, it is time to refine the non-functional software architecture. In order to do this, some points must be considered:

- non-functional software architectures are (in practice) designed incrementally yielding a hierarchy (see Figure 4). As each design decision is made, the hierarchy is transformed to reflect the current state of the design. If all transformations of the hierarchy preserve its correctness, and the starting point is a trivially correct refinement-free hierarchy, then the completed hierarchy that eventually results is guaranteed to be correct by construction [32];
- an architecture transformation rule is correct if and only if, whenever it is applied to a correct hierarchy, a correct hierarchy results [32], i.e., correct transformation rules preserve hierarchy correctness;

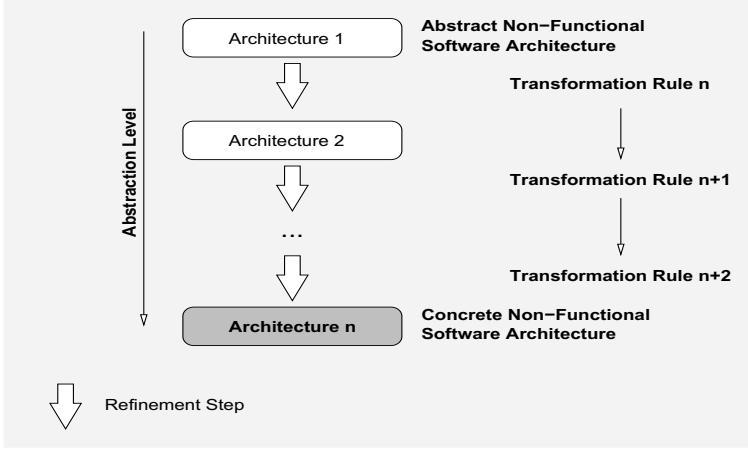

Figure 4: Refinement Overview

- Moriconi [27] defines that architectural elements (components, connectors and interfaces) are refined by replacing, decomposing, aggregating and removing them.

Considering these points, we introduce in a semi-formal manner some transformation rules that can be adopted to refine the non-functional software architecture. Essentially, these transformation rules place together non-functional properties and architectural elements, i.e., Components, Connectors, Architecture, NF-Attributes, NF-Actions and NF-Statements that are present in the non-functional software architecture.

The following sections present the proposed transformation rules, which are divided into four main classes: decomposition, aggregation, elimination and replacement rules. The description of each rule contains four parts that includes: an informal introduction of the transformation rule (**Informal Description**); a justification that indicates which concrete situation is envisaged for it (**Motivation**); a semi-formal description of the transformation including basic concepts, the input of the transformation, the expected result of the transformation (**Semi-Formal Description**) and requirements that must be satisfied before the transformation can be applied (**Rule Requirements**); the transformation itself (**Solution**); and an example whose primary purpose is to clarify the transformation (**Example**).

3.1 Decomposition Rules

Decomposition rules are applied for decomposing non-functional and architectural elements into more concrete ones.

3.1.1 Decomposition of NF-Attributes

Informal Description

This rule decomposes an NF-Attribute into more primitive NF-Attributes. The architectural elements that have assigned constraints on the NF-Attribute (NF-Statement) comes to share the new primitive NF-Attributes.

Motivation

The usual early description of NF-Attributes simply defines general constraints on them without any additional information about how to achieve the desired constraint. This fact is also normal as the notion of NF-Attributes differs from one domain to another. For instance, at a first stage, *good performance* means to do something as fast as possible, but they have different interpretations according to the part of the application it is assigned to. A server (component) with a *good performance* may be interpreted as one that processes as many transactions as possible and a middleware (connector) [40] with *good performance* typically defines one able to transmit as many message as possible between two or more components (throughput).

Semi-formal Description

Auxiliary Concepts

$sta = (S_{sta}, c, p)$ is a NF-Statement, where
$c = \{..., (att, c_{att}), ...\}$ defines the constraints imposed over the NF-Attributes,
$p = \{..., (att, p_{att}), ...\}$ defines the priority of the NF-Attributes,
$att = (S_{att}, cont)$ is a NF-Attribute ($att \in \mathsf{attributes}.sta$),
$nf\_comp = comp \succ sta$ is the NF-Component,
$Actions$ is the set of NF-Actions.

Input
att is an NF-Attribute.
Output
att' and *att''* are the NF-Attributes that result from the decomposition of *att*.

Rule Requirements

- the NF-Attribute being refined (*att*) must belong to the set of NF-Attributes (attributes.*sta*) that are constrained by the NF-Statement (*sta*), i.e., *att* ∈ attributes.*sta*,
- no NF-Action (*act*) may have already been defined that implements or affects the NF-Attribute being refined, i.e., ∀ *act* ∈ *Actions*, *att* ∉ affectedAttributes.*act* and *att* ∉ implementedAttributes.*act*.

Solution ($\mathcal{T}_1$)
att $\longrightarrow_{\mathcal{T}_1}$ (*att'*,*att''*), where

- *att'* belongs to the set of NF-Attributes constrained by the NF-Statement *sta*, *att'* has the same kind of constraint and priority of the original NF-Attribute (*att*) and the set of NF-Attributes that make up *att'* is empty (it has been just defined), i.e., *att'* ∈ constrainedAttributes.*sta*, (*att'*,c_{att}) ∈ kindOfConstraint.*sta*, (*att'*,p_{att}) ∈ priority.*sta* and attributes.*att'* = ∅,
- *att''* belongs to the set of NF-Attributes constrained by the NF-Statement *sta*, *att''* has the same kind of constraint and priority of the original NF-Attribute (*att*) and the set of NF-Attributes that make up *att''* is empty (it has been just defined), i.e., *att''* ∈ constrainedAttributes.*sta*, (*att''*,c_{att}) ∈ kindOfConstraint.*sta*, (*att''*,p_{att}) ∈ priority.*sta* and attributes.*att''* = ∅.

Example
performance = ({},*none*) is the NF-Attribute to be refined,
good_performance = ({*performance*},{(*performance*,*strong*)},{(*performance*,*high*)}) is the NF-Statement,
nf_server = *server* ≻ *good_performance* is a NF-Component,
performance $\longrightarrow_{\mathcal{T}_1}$ (*space_performance*, *time_performance*) is the application of the transformation rule that decomposes the NF-Attribute *performance* into *space_performance* and *time_performance*, where

- *space_performance* = ({},*none*) is the first NF-Attribute that results of the transformation,
- *time_performance* = ({},*none*) is the second NF-Attribute that result of the transformation,
- *good_performance* = ({*performance*, *space_performance*, *time_performance*}, {(*space_performance*, *strong*), (*time_performance*, *strong*)}, {(*space_performance*, *high*), (*time_performance*, *high*)}) is the NF-Statement after the transformation of a NF-Attribute that is constrained by *good_performance*.

3.1.2 Decomposition of NF-Components

Informal Description
This rule decomposes a Component into two other ones preserving the NF-Statement associated with the original component.
Motivation
NF-Components are usually defined abstractly, while they are subsequently refined into more concrete components. However, this particular decomposition takes into account that NF-Statements are present and must be respected. This means that the decomposition yields NF-Components with similar NF-Statements as the original component.
Semi-formal Description
Auxiliary Concepts
$nf\_comp = comp \succ sta_{comp}$ is an NF-Component,
$nf\_arc = arc \succ sta_{arc}$ is an NF-Architecture.
Input
nf_comp is an NF-Component.
Output
nf_comp' and nf_comp'' are the NF-Components that from the decomposition of nf_comp.
Rule Requirements
none.
Solution ($\mathcal{T}_2$)
$nf\_comp \longrightarrow_{\mathcal{T}_2} nf\_comp'$, nf_comp'', where

- nf_comp' = $comp$' $\succ$ $sta_{comp'}$ is the first component of the decomposition, which has the same NF-Statement as the original component, i.e., $sta_{comp'} = sta_{comp}$,
- nf_comp" = $comp$" $\succ$ $sta_{comp''}$ is the second component of the decomposition, which also has the same NF-Statement as the original component, i.e., $sta_{comp''} = sta_{comp}$.

Example *nf_server* = *server* $\succ$ *secure* is the NF-Component to be refined. It has the NF-Statement *secure* assigned to,

nf_server $\longrightarrow_{\mathcal{T}_2}$ (*nf_requestReceiver*,*nf_requestResponse*) is the application of the transformation rule that decomposes the component *nf_server* into *nf_requestReceiver* and *nf_requestResponse*, where

- *nf_requestReceiver* = *requestReceiver* $\succ$ *secure* is the new element that make up the *nf_server*, which also has the same NF-Statement (*secure*) as the *nf_server*,
- *nf_requestResponse* = *requestResponse* $\succ$ *secure* is another new component that make up the *nf_server*, which also has the same NF-Statement (*secure*) as the *nf_server*.

3.1.3 Decomposition of NF-Actions

Informal Description

This rule decomposes NF-Actions that affect or implement NF-Attributes into NF-Actions closer to actual implementation elements.

Motivation

The decomposition of an NF-Action is important as it represents an advance on the implementation elements that realise or affect NF-Attributes.

Semi-formal Description

Auxiliary Concepts

att = (S_{att},*cont*) is a NF-Attribute,
act = (A_{act},I_{act},e_{act}) is a NF-Action.

Input

act is the NF-Action to be decomposed.

Output

act' and *act*" are the NF-Actions that result from the decomposition of *act*.

Rule Requirements

none.

Solution ($\mathcal{T}_3$)

act $\longrightarrow_{\mathcal{T}_3}$ *act*', *act*", where

- act' = ($A_{act'}$,$I_{act'}$,$e_{act'}$) is the new NF-Action that results from the decomposition of *act*, *act*' also affects/implements at least (additional NF-Attributes may also be affected by the new NF-Action) the NF-Attibutes affected/implemented by *act*, i.e., affectedAttributes.*act* $\subseteq$ affectedAttributes.*act*' and implementedAttributes.*ac* $\subseteq$ implementedAttributes.*act*',
- *act*" = ($A_{act''}$,$I_{act''}$,$e_{act''}$) is the new NF-Action that results from the decomposition of *act*, *act*" also affects/implements at least (additional NF-Attributes may also be affected by the new NF-Action) the NF-Attibutes affected/implemented by *act*, i.e., affectedAttributes.*act* $\subseteq$ affectedAttributes.*act*" and implementedAttributes.*a* $\subseteq$ implementedAttributes.*act*".

Example

authoriseAccess = ({},{*confidentiality*},{}) is the NF-Action to be decomposed,

authoriseAccess $\longrightarrow_{\mathcal{T}_3}$ (*identifyUser*,*authenticateUser*) is the application of the transformation rule that decomposes *authoriseAccess* into *identifyUser* and *authenticateUser*, where

- *identifyUser* = ({},{*confidentiality*},{}) is the first NF-Action that results from the decomposition of *authoriseAccess*,
- *authenticateUser* = ({},{*confidentiality*},{}) is the second NF-Action that results from the decomposition of *authoriseAccess*.

3.2 Aggregation Rules

Aggregation rules are defined in order to enable architectural elements to be put together.

3.2.1 Aggregation of NF-Components

Informal Description

This rule aggregates NF-Components in order to compose them into a more complete NF-Component.

Motivation

The necessity to compose may be motivated by the necessity to eliminate communication between two components. For instance, if two components communicate a lot, it may be interesting put both in a single component. However, as in the decomposition, non-functional properties must be preserved.

Semi-formal Description

Auxiliary Concepts

$sta = (S_{sta}, c, p)$ is a NF-Statement,

$nf-comp = comp \succ sta$ is a component that has assigned the NF-Statement sta.

Input

nf_comp' $= comp' \succ sta'$ is first of the component to be aggregated,

nf_comp" $= comp" \succ sta"$ is the second component to be aggregated.

Output

nf_comp is the component that results from the aggregation.

Rule Requirements

sta' and sta'' are compatible in the sense that they do not cause a conflict as stated in Definition 2.4.

Solution ($\mathcal{T}_4$)

nf_comp',$nf\_comp"$ $\longrightarrow_{\mathcal{T}_4}$ nf_comp, where

- $nf\_comp = comp \succ sta$ is the new component that has assigned the NF-Statement sta, where sta is the union of the two NF-Statements of the original components (nf_comp' and nf_comp"), i.e., constrainedAttributes.sta = (constrainedAttributes.sta' $\cup$ constrainedAttributes.sta"), kindOfConstraint.sta = (kindOfConstraint.sta' $\cup$ kindOfConstraint.sta") and priority.sta = (priority.sta' $\cup$ priority.sta").

Example

$nf\_requestReceiver = request\_receiver \succ secure$ and $nf\_replyTransmitter = reply\_transmitter \succ secure$ are the components to be aggregated,

($nf\_requestReceiver$,$nf\_replyTransmitter$) $\longrightarrow_{\mathcal{T}_4}$ nf_server is the application of the aggregation rule, where

- $nf\_server = server \succ secure$ is the new server that results from the aggregation of the components $nf\_requestReceiver$ and $nf\_replyTransmitter$.

3.3 Elimination Rules

Elimination rules are used to remove parts of the design that are not necessary in the software development. Reasons for the elimination comes from the fact that other components incorporate the functionality that is implemented by the component.

3.3.1 Elimination of NF-Components

Informal Description

This rule eliminates an NF-Component of the software architecture.

Motivation

Necessity to remove unnecessary parts of the software architecture during the development.

Semi-formal Description

Auxiliary Concepts

$nf\_comp = comp \succ sta_{comp}$ is a component that has assigned the NF-Statement sta_{comp},

$nf\_arc = arc \succ sta_{arc}$ is the software architecture that has assigned the NF-Statment sta_{arc}.

Input

nf_arc is the NF-Architecture to be altered,

nf_comp is the NF-Component to be removed.

Output

nf_arc' is the NF-Architecture with the NF-Component removed.

Rule Requirements

none.

Solution ($\longrightarrow_{\mathcal{T}_5}$)

$nf\_arc \longrightarrow_{\mathcal{T}_5} nf\_arc$', where nf_comp_h is *removed*

- $nf\_arc = (\{nf\_comp_1, ..., nf\_comp_h, ..., nf\_comp_n\}, \{nf\_conn_1, ..., nf\_conn_m\}, \{(nf\_comp_i, nf\_conn_j, nf\_comp_{i+1}), ..., (nf\_comp_k, nf\_conn_l, nf\_comp_{k+1})\} \succ sta_{arc}$ is the new software architecture in which the component nf_comp_h has been removed.

Example

$nf\_client\_server\_arc = (\{nf\_client, nf\_main\_server, nf\_secondary\_server\}, \{nf\_request\_reply\}, \{(nf\_client$ $nf\_request\_reply, main\_server), (nf\_client, nf\_request\_reply, nf\_secondary\_server)\}) \succ secure$ is the software architecture and $secondary\_server$ is the component to be removed,

$(nf\_client\_server\_arc, nf\_secondary\_server) \longrightarrow_{\mathcal{T}_5} nf\_client\_server\_arc'$ is the application of the elimination rule, where

$nf\_client\_server\_arc' = \{nf\_client, main\_server\}, \{nf\_request\_reply\}, \{(nf\_client, nf\_request\_reply, nf\_main\_server)\}) \succ secure$ is the new architecture without $nf\_secondary\_server$.

3.4 Replacement Rules

Replacement rules are applied to substitute a component with another, more refined, one. Basically, the new component must satisfy the constraints imposed by the NFRs.

3.4.1 Replacement of NF-Components

Informal Description

This rule replaces the NF-Component with another one.

Motivation

The motivation to replace a component appears when a new solution (approach), better than the previous one, is found to perform a specific task.

Semi-formal Description

Auxiliary Concepts

$nf\_arc = arc \succ sta$ is a software architecture that has assigned the NF-Statement sta.

Input

nf_arc is the NF-Architecture to be altered,

nf_comp, nf_comp' are the old and new components, respectively.

Output

nf_arc' is the NF-Architecture after the replacement of a component.

Rule Requirements

The NF-Statements (sta_h and sta_h') assigned to the replaced component must be preserved in the updated architecture.

Solution ($\mathcal{T}_6$)

$nf\_arc, nf\_comp \longrightarrow_{\mathcal{T}_6} nf\_arc$' is the application of the replacement rule (nf_comp_h is replaced by nf_comp_h'), where

- $nf\_arc = (\{nf\_comp_1, ..., nf\_comp_h, ..., nf\_comp_n\}, \{nf\_conn_1, ..., nf\_conn_m\}, \{(nf\_comp_i, nf\_conn_j, nf\_comp_{i+1}), ..., (nf\_comp_k, nf\_conn_l, nf\_comp_{k+1})\} \succ sta_{arc}$ is the original architecture.
- $nf\_arc' = \{nf\_comp_1, ..., nf\_comp_h, ..., nf\_comp_n\}, \{nf\_conn_1, ..., nf\_conn_m\}, \{(nf\_comp_i, nf\_conn_j, nf\_comp_{i+1}), ..., (nf\_comp_k, nf\_conn_l, nf\_comp_{k+1})\} \succ sta_{arc}$. is the new architecture with nf_comp_h has been replaced by nf_comp_h'.

Example

$nf\_client\_server\_arc = (\{nf\_client, nf\_server\}, \{nf\_request\_reply\}, \{(nf\_client, nf\_request\_reply, nf\_server)\}) \succ secure$ is the initial software architecture,

$nf\_server = server \succ secure$ is the component to be replaced,

$nf\_powerful\_server = server \succ secure\_and\_fast$ is the new component,

$(nf\_client\_server\_arc, nf\_server, nf\_server\_powerful)$ $\mathcal{T}_6$ $nf\_client\_server\_arc$' is the application of the replacement rule, where the NF-Statement of the original component ($secure$) is preserved in the new component $nf\_server\_powerful$ ($secure\_and\_fast$).

4 Case Study: An Appointment System over an Object-Oriented Middleware

The Appointment System is a client-server application that carries out distributed appointment scheduling and executes over an object-oriented middleware (OOM) [16]. The clients and server communicate through an OOM that provides synchronous communication and distributed services such as *transaction* and *security*. The server must provide operations that can be used to add and remove an appointment, while it also defines an operation that returns the current time schedule. The system must be *secure* and *fast*, while all operations must be executed respecting the ACID transactional properties *atomicity*, *consistency*, *isolation* and *durability*. Additionally, the server may be dynamically replaced at any time during its execution.

4.1 Non-Functional Aspects

According to the Appointment System's description, three NF-Attributes are involved in its development: *ACID* for transactional properties, *performance* and *security*. They are initially defined as follows:

$$\begin{aligned} ACID &= (\{\}, none), \\ performance &= (\{\}, none), \\ security &= (\{\}, none). \end{aligned}$$

The constraints defined in the system description ("the system must be *secure* and *fast*, while all operations must be executed respecting the ACID transactional properties"), are modelled through the NF-Statements $Acid\_Property$, $Fast$ and $Secure$:

$$\begin{aligned} Acid\_Property &= (\{ACID\}, \{(ACID, present)\}, \{(ACID, high)\}), \\ Fast &= (\{performance\}, \{(performance, strong)\}, \{(performance, medium)\}), \\ Secure &= (\{security\}, \{(security, strong)\}, \{(security, high)\}), \\ Fast\_Secure &= (\{performance, security\}, \\ &\quad \{(performance, strong), (security, strong)\}, \\ &\quad \{(performance, medium), (security, high)\}). \end{aligned}$$

4.2 Architectural Aspects

The abstract software architecture derived from the Appointment System description is shown in Figure 5. In order to define the architecture, we adopted the client-server style, which means that only client and server components are allowed. Three kinds of components, namely client (*Client_1*, *Client_2*), server (*Server*) and database (*Database*), and two kinds of connectors (*Middleware*,*ODBC*) make up the software architecture. Requests originating from clients are handled by the server, which accesses the database to store or retrieve the information. The communication between clients and the server is performed through the connector *Middleware*, while the connector *ODBC* is used to communicate the server and the database.

Formally,

$$\begin{aligned} Client\_1 &= (int_{client\_1}), \\ Client\_2 &= (int_{client\_2}), \\ Server &= (int_{server}), \\ Database &= (int_{database}), \\ Middleware &= (int_{c\_1}), \\ ODBC &= (int_{c\_2}), \\ AppointmentSystem &= (\{Client\_1, Client\_2, Server, Database\}, \\ &\quad \{Middleware, ODBC\}, \\ &\quad \{(Client\_1, Middleware, Server), \\ &\quad (Client\_2, Middleware, Server), \\ &\quad (Server, ODBC, Database)\}). \end{aligned}$$

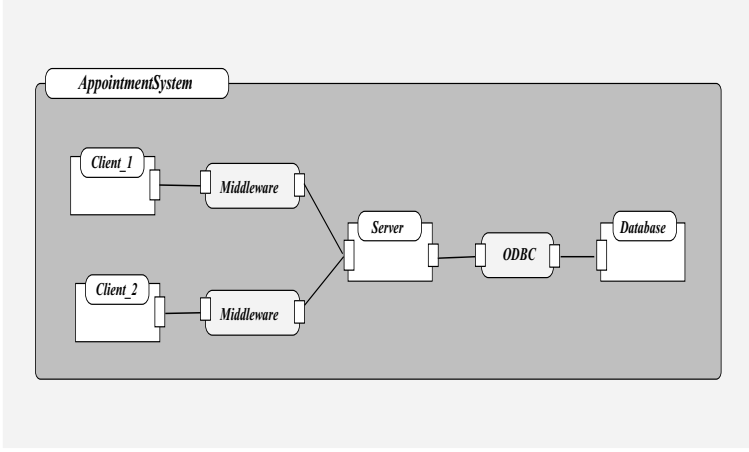

Figure 5: Appointment System Abstract Architecture

4.3 Integration of Architectural and Non-Functional Aspects

Considering the software architecture presented in Figure 5 and the non-functional elements defined previously, the NF-Statement $Good\_Performance\_Secure$ is associated with the *Server*. This decision is motivated by the fact that it is the main processing element in the software architecture. The clients must also be *secure*, but not necessarily *fast*, as they are not central processing elements. Meanwhile, to the *Database* is assigned the transactional properties $ACID\_Property$ and *Secure*.

Formally,

$$\begin{aligned} NF\_AppointmentSystem &= AppointmentSystem \prec Secure, \\ NF\_Server &= Server \prec Fast\_Secure, \\ NF\_Database &= Database \prec ACID\_Property \wedge Secure. \end{aligned}$$

According to the integration rules, the NF-Statement *Secure* is defined for every element of the software architecture. Hence,

$$\begin{aligned} NF\_Client\_1 &= Client\_1 \prec Secure, \\ NF\_Client\_2 &= Client\_2 \prec Secure. \end{aligned}$$

In terms of connectors, we consider that they must be *secure*. Hence,

$$\begin{aligned} NF\_Middleware &= Middleware \prec Secure, \\ NF\_ODBC &= ODBC \prec Secure. \end{aligned}$$

In this particular case, an object-oriented middleware that provides the security service, e.g., CORBA (Commom Object Request Broker Architecture) [31] may be adopted in the implementation of the appointment system.

4.4 Refinement of the Abstract Software Architecture

The initial specification of the NF-Attributes *performance* and *security* and the NF-Architecture $NF\_AppointmentSer$ can be refined according the rules presented in Section 3.

The first rule used ($\longrightarrow_{\mathcal{T}_1}$) decomposes the NF-Attributes *performance*, *ACID* and *security* into more primitive NF-Attributes:

performance $\longrightarrow_{\mathcal{T}_1}$ (*space_performance,time_performance*),
ACID $\longrightarrow_{\mathcal{T}_1}$ (*atomicity,consistency,isolation,durability*),
security$\longrightarrow_{\mathcal{T}_1}$ *(data_confidentiality,integrity).*

The next refinement consist of decomposing the NF-Component NF_Server into two components, one for receiving the requests from a NF_Client and another for sending the result (received from $NF\_Database$) to the caller. In this case, the rule $\longrightarrow_{\mathcal{T}_2}$ is applied as shown in the following:

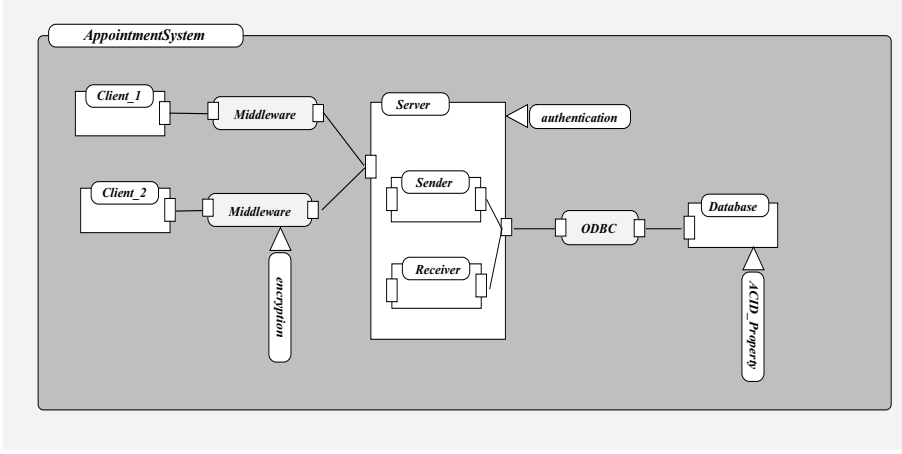

Figure 6: Appointment System Concrete Software Architecture

$NF\_Server \longrightarrow_{\mathcal{T}_2} (NF\_Receiver, NF\_Sender)$,
where, $NF\_Receiver$ and NF_Sender also have the same NF-Statement assigned to the NF_Server. Hence,

$$\begin{aligned} NF\_Receiver &= Receiver \prec Fast\_Secure, \\ NF\_Sender &= Sender \prec Fast\_Secure. \end{aligned}$$

NF-Actions related to *security* are shown in the following:

$$\begin{aligned} accessControl &= (\{\}, \{security\}, \{\}), \\ intrusionDetection &= (\{\}, \{security\}, \{\}), \\ authentication &= (\{\}, \{security\}, \{\}), \\ encryption &= (\{\}, \{security\}, \{(performance, against, -1)\}). \end{aligned}$$

In order to refine the NF-Actions *authentication* and *intrusionDetection*, we apply the rule $\longrightarrow_{\mathcal{T}_3}$ that decomposes NF-Actions:
$authentication \longrightarrow_{\mathcal{T}_3} (userIdRequest, passwordRequest)$,
$intrusionDetection \longrightarrow_{\mathcal{T}_3} (verifyAccessFromStrangeSites,$
$checkAskingForManyQuestions,$
$checkPasswordAttempts)$,
$encryption \longrightarrow_{\mathcal{T}_3} (algorithmRSA, use128bits)$
After the refinement, we have the concrete software architecture shown in Figure 6 .

5 Related Work

The related work have been divided into three categories that are most closely related to the subject in hand: the treatment of non-functional requirements, the formalisation of non-functional requirements and the checking of software properties during runtime changes. The treatment of non-functional properties includes researches related to software development such as the NFR Framework, while also present those that concentrate on the description of non-functional properties once the software is already built such as the NoFun notation. We will further discuss the formalisation of non-functional aspects.

Efforts for treating NFRs begin by taking them in account during software development. At this time, some approaches try to define how to incorporate these requirements into the software development until their realisation. Our presentation begins with the introduction of the NFR framework.

The NFR Framework The NFR Framework [4, 28, 8] concentrates on the explicit representation and analysis of NFRs during software development. In order to carry out this task, the framework first gives the meaning of NFRs and provides a set of abstractions for modelling them. It then defines how to record the design process considering

design decisions that can be taken for accomplishing the NFRs. It includes the definition of the relationships between the NFRs and their refinement. Finally, the framework performs an evaluation procedure that decides whether each design decision is good enough, i.e., whether it meets a softgoal sufficiently or not. This framework has been used to deal with NFRs such as *accuracy* [4], *security* [5], *performance* [29] (see Figure **??**) and *modifiability* [3]. Some of these requirements have been evaluated from a more practical view in Chung [6], where the framework is used to treat NFRs in credit card, health insurance and government systems. In addition, the framework has been used to show how the record of the treatment of NFRs can help to deal with changes in the software[2] [7].

The Injectors Approach The Object Infrastructure Project (OIP) presented in Filman [11, 12] refers to NFRs as *ilities*. *Ilities* (NFRs) are manifestations of properly defined and implemented requirements. Unlike previous approaches, this one concentrates on the realisation phase only. It considers that certain *ilities* such as *security*, *reliability*, *manageability* and *quality of service* are achieved through controlling both the communication between components and significant events of an object's life-cycle.

According to this approach, *security* is primarily viewed as the combination of access control, intrusion detection, authentication and encryption; *manageability* is implemented by generating events related to performance measurement, accounting, failure analysis and intrusion detection; *reliability* is simply realised through the implementation of replication; and *quality of service* is implemented by calling system quality of service primitives, using side-door mechanisms to efficiently transport large quantities of data, using queue control to identify the most worthwhile thing to do next and by choosing among multiple ways of problem solving.

In practice, this strategy works intercepting and manipulating the communication between components. The interception is performed by objects named *injectors,* which take responsibility to provide (combined with others or not) the required ility. The injectors are inserted into functional components and may be assigned to a single operation or to an entire component. This is performed by giving a functional code, ility specification and the ility service implementation (injector), which are weaved together into the actual system code. In order to express these elements, the Pragma Language [13] is provided. A specification in Pragma consists of the identification of required *ilities* and for each ility the actions (e.g., a set of algorithms) that can be taken to achieve it.

The Aster Approach In a similar way to the previous approach, the Aster framework [18, 19] also concentrates on the implementation of NFRs. This task is carried out by customising distributed runtime systems according to NFRs required by distributed applications. In the heart of the Aster framework is the notion that satisfying application's requirements relies on the use of an appropriate or customised distributed runtime system[3].

The runtime-distributed system is a middleware used to communicate components during runtime. The customised middleware best maps an application with specific requirements (NFRs) onto an execution platform with specific constraints. The basic task of this approach consists of matching NFRs required by the application against ones provided by the composition of available software components and middleware. The outcome of this matching is a customised middleware, made up of the selected components and base connectors, which implements the NFRs required by the application.

In order to carry out the customisation, the first task is to describe the application, together with its NFRs. An architecture-based approach is adopted, in which a notation, namely Aster language, is proposed for describing software architectures. This language, however, has extra skills for expressing NFRs. The second task is responsible for taking this description and tries to find software components (stored in a database) that can be composed for matching the NFRs defined in the software architecture. The last task takes the selected software components and puts them together to make up the middleware in which the application will execute.

The Aster framework has been used to implement transactional properties [41, 42, 39], *security* [1, 2], *fault-tolerance* [38, 21], *dynamic* property that express the possibility of the dynamic binding of a client with a file server [20], *reliability* [43], *efficiency* [14] and *dependability* [22].

The Aspect-Oriented Approach Another approach to dealing with NFRs adopts Aspect-Oriented Programming (AOP) principles. According to this approach, an application is decomposed into functional components and aspects, where aspects model concerns that occur throughout the entire application and therefore cross functional element[4] boundaries, e.g., *dependability, real-time performance* and *security* [24]. Each aspect is programmed in a particular language suitable for expressing it and then distributed into the proper places throughout the application.

Based on this paradigm, Loyall [26] proposes the QuO framework (Quality Objects) for developing distributed systems (client-server applications) with QoS (Quality of Service) requirements. In order to do this, two aspect languages are proposed for specifying QoS requirements, together with a runtime support for their monitoring, control

[2] Adding or modifying NFRs or changes in the design.
[3] Also referred to *software bus* [19].
[4] Procedures, objects, modules, etc.

and adaptation to changing levels. QoS requirements are specified through contracts described in CDL (Contract Description Language) and SDL (Structure Description Language). The specification of QoS requirements consist of defining predicates on system conditions, e.g., for a *high availability* the predicate `MeasuredNumberReplicas > 1` must be true. The contracts contain the QoS desired by clients, the level of QoS a server expects to provide, operating regions representing possible states of QoS (system conditions), transitions between the states and actions that have to be taken when the level of QoS changes.

Other Approaches The approach proposed by Justo [23] describes a Java-based framework to support the definition and control of NFRs of distributed component architectures. The essence of the proposed approach is to demonstrate how the existing concepts of components services and architectures can be extended to support the NFRs description and control at run-time.

Another approach to integrating NFRs into software architecture is proposed by Robben [33]. In this approach, NFRs such as *reliability* and *security* are individually realised by single components and properly integrated into software architectures. NFRs have also been addressed in the development of knowledge-based systems [25], distributed systems built from mobile agents [14], network management systems [30], COTS selection in a component-based software [34] and Avionics Control System [10].

6 Conclusion

This paper has presented a semi-formal approach for reasoning and refining non-functional requirements. Essent ially, it proposed a set of abstractions to reason about non-functional requirements together a set of refinement rules for refining non-functional requirements during the software development process. Non-functional properties are expressed through three abstractions, NF-Attribute, NF-Statement and NF-Action. These abstractions are incorporated into software architecture element and then refined.

The primary contribution of this paper is the explicit treatment of NFRs during the development of software systems. The set of actions taken to perform this explicit treatment makes contributions in three main points: the general foundations of non-functional properties, the integration of these properties with functional elements and the their refinement.

There is a growing body of work on understanding the foundations of non-functional aspects of software systems. Using our abstractions, we clearly define how to reason about these aspects as an initial step towards a better comprehension of what non-functionality means. Their explicit relationship enables us to reason about their correlation and conflicts. Additionally, our approach is not related to any particular non-functional property. This fact allows us to reason about non-functional properties in general, without trying to focus on particular characteristics that can blur their treatment.

In terms of refinement, refinement rules help to understand and trace the trajectory of non-functional properties during software development. Our contribution to this particular point concentrates on the exploitation of the refinement of software architecture, taking into account the constraints imposed by the non-functional aspects. This approach has been adopted in the refinement of software radios [35] and an appointment system.

This approach is an initial contribution to the complex task of the treatment of non-functional requirements. It serves as a general guideline on how to consider NFRs and creates the possibility of some interesting future work: the definition of the formal semantics of the proposed notations; tools that enable the developer to check properties of the non-functional properties, e.g., the automatic verification of conflicts and correlation between them; and the formalisation of the notion of satisfaction that can enable us to precisely define the degree of satisfaction of the non-functional requirements in the final product.

References

[1] C. Bidan and Valèrie Issarny. Security Benefits from Software Architecture. In *Second International Conference on Coordination Models and Languages*, pages 64–80, Berlin, Germany, September 1997.

[2] Christophe Bidan and Valèrie Issarny. Dealing with Multi-Policy Security in Large Open Distributed Systems. In Jean-Jacques Quisquater et. al, editor, *Fifth European Symposium on Research in Computer Security (ESORICS'98)*, volume 1485 of *Lecture Notes in Computer Science*, pages 51–66, Louvain-la-Neuve, Belgium, September 1998.

[3] L. Chung, D. Gross, and E. Yu. Architectural Design to Meet Stakeholder Requirements. In *First Working IFIP Conference on Software Architecture (WICSA1)*, pages 545–564, San Antonio, Texas, USA, February 1999.

[4] Lawrence Chung. Representation and Utilization of Non-Functional Requirements for Information System Design. In *Third International Conference on Advanced Information System Engineering (CAiSE'91)*, pages 5–30, Trondheim, Norway, May 1991.

[5] Lawrence Chung. Dealing with Security Requirements During the Development of Information Systems. In *Fifth International Conference on Advanced Information System Engineering (CAiSE'93)*, pages 234–251, Paris, France, June 1993.

[6] Lawrence Chung and Brian A. Nixon. Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach. In *Seventeenth International Conference on Software Engineering (ICSE'95)*, pages 25–37, Seattle, USA, April 1995.

[7] Lawrence Chung, Brian A. Nixon, and Eric Yu. Using Non-Functional Requirements to Systematically Support Change. In *Second IEEE International Symposium on Requirements Engineering*, pages 132–139, York, England, March 1995.

[8] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering.* Kluwer Academic Publishers, 2000.

[9] Paul C. Clements. Coming Attractions in Software Architecture. In *Joint Workshop on Parallel and Distributed Real-Time Systems (WPDRTS/OORTS'97)*, Geneva, Switzerland, April 1997.

[10] Bruno Dutertre and Victoria Starvridou. Formal Requirements Analysis of an Avionics Control System. *IEEE Transactions on Software Engineering*, 23(5):267–278, May 1997.

[11] R. E. Filman. Achieving Ilities. In *Workshop on Compositional Software Architectures*, Monterey, California, USA, January 1998.

[12] R. E. Filman. Injecting Management. In *International Workshop on Component-Based Software Engineering*, Kyoto, Japan, April 1998.

[13] R. E. Filman, Stu Barret, Diana Lee, and Ted Linden. Inserting Illities by Controlling Communications. *Communications of the ACM*, August 1999.

[14] Pascal Fradet, Valérie Issarny, and Siegfried Rouvrais. Analyzing Non-Functional Properties of Mobile Agents. In *Fundamental Approaches to Software Engineering (FASE'00)*, volume 1783 of *Lecture Notes in Computer science*, pages 319–328, Berlin, Germany, March 2000.

[15] Aditya K. Ghose. Managing Requirements Evolution: Formal Support for Functional and Non-Functional Requirements. In *International Workshop on Principles of Software Evolution (IWPSE'99)*, pages 118–124, Fukuoka, Japan, July 1999.

[16] Michi Henning. A New Approach to Object-Oriented MIddleware. *IEEE Internet Computing*, 8(1):66–75, February 2004.

[17] IEEE/ANSI. *830-1998 Recommended Practice for Software Requirements Specifications*, 1998.

[18] Valèrie Issarny and Christophe Bidan. Aster: A CORBA-Based Software Interconnection System Supporting Distributed System Customization. In *Third International Conference on Configurable Distributed Systems (ICCDS'96)*, pages 194–201, Annapolis, Maryland, USA, May 1996.

[19] Valèrie Issarny and Christophe Bidan. Aster: A Framework for Sound Customization of Distributed Runtime Systems. In *Sixteenth International Conference on Distributed Computing Systems*, pages 586–593, Hong Kong, May 1996.

[20] Valèrie Issarny, Christophe Bidan, and Titos Saridakis. Achieving Middleware Customization in a Configuration-Based Development Environment: Experience with the Aster Prototype. In *Fourth International Conference on Configurable Distributed Systems (ICCDS'98)*, pages 207–214, Annapolis, Maryland, USA, 1998.

[21] Valèrie Issarny, Christophe Bidan, and Titos Saridakis. Characterizing Coordination Architectures According to their Non-Functional Execution Properties. In *Third-First Annual Hawaii International Conference on System Sciences (HICSS-31)*, pages 275–283, Hawaii, USA, January 1998.

[22] Valèrie Issarny, Titos Saridakis, and Apostolos Zarras. Multi-View Description of Software Architectures. In *Third International Workshop on Software Architecture*, pages 81–84, Orlando, Flórida, USA, November 1998.

[23] George R. R. Justo and Ahmed Saleh. Non-Functional Integration and Coordination of Distributed Component Service. In *Sixth European Conference on Software Maintenance and Reengineering*, Budapest, Hungary, March 2002.

[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, and Jean-Marc Loingtier. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, Finland, June 1997.

[25] D. Landes. Addressing Non-Functional Requirements in the Development of Knowledge-Based Systems. In *First International Workshop on Requirements Engineering: Foundation of Software Quality*, pages 64–70, Utrecht, Netherlands, June 1994.

[26] Joseph P. Loyall, David E. Bakken, Richard E. Schants, John A. Zinky, David A. Karr, Rodrigo Vanegas, and Kenneth R. Anderson. QoS Aspect Language and their Runtime Integration. In *Fourth Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, volume 1511 of *Lecture Notes in Computer Science*, Pittsburgh,Pennsylvania,USA, May 1998. Springer-Verlag.

[27] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[28] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and Using Non-Functional Requirements: A Process-Oriented Approach. *IEEE Transaction of Software Engineering*, 18(6):483–497, June 1992.

[29] Brian A. Nixon. Representing and Using Performance Requirements During the Development of Information Systems. In *Fourth International Conference on Extending Database Technology (EDBT'94)*, pages 187–200, Cambridge, United Kingdom, March 1994.

[30] Natsuko Noda and Tomoji Kishi. An Architectural Approach to Performance Issues: From Experience in the Development of Network Management Systems. In *First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, Texas, USA, February 1999. Position Paper.

[31] OMG. *Common Object Request Broker Architecture: Core Specification (CORBA 3.0)*, December 2002.

[32] R. A. Riemeschneider. Correct Transformation Rules for Incremental Development of Architecture Hierarchies. Technical Report Paper DSA-98-01, Dependable System Architecture Group Working, SRI International, 1998.

[33] Bert Robben, Matthijs, Wouter Joosen, Bart Vanhaute, and Pierre Verbaeten. Components for Non-Functional Requirements. In *Third International Workshop on Component-Oriented Programming*, Brussels, Belgium, July 1998. Position Paper.

[34] Nelson S. Rosa, Carina F. Alves, Paulo R. F. Cunha, Jaelson F. B. Castro, and George R. R. Justo. Using Non-Functional Requirements to Select Components: A Formal Approach. In *Fourth Ibero-American Workshop on Software Engineering and Software Environment*, San Jose, Costa Rica, April 2001.

[35] Nelson S. Rosa, Paulo R. F. Cunha, and George R. R. Justo. Expressing Real-Time Performance in Software Radios. In *Twenty-Second IEEE Real-Time Systems Symposium*, London, UK, December 2001.

[36] Nelson S. Rosa, George R. R. Justo, and Paulo R. F. Cunha. Incorporating Non-Functional Requirements into Software Architecture. In José Rolim et. al, editor, *Fifth International Workshop on Formal Methods for Parallel Programming: Theory and Applications*, volume 1800 of *Lecture Notes in Computer Science*, pages 1009 – 1018, Cancun, Mexico, May 2000.

[37] Nelson S. Rosa, George R. R. Justo, and Paulo R. F. Cunha. A Framework for Building Non-Functional Software Architectures. In *16th ACM Symposium on Applied Computing*, pages 141–147, Las Vegas, USA, March 2001.

[38] Titos Saridakis and Valèrie Issarny. Fault Tolerant Software Architectures. Technical Report RR-3350, INRIA, January 1998. 17 p.

[39] Titos Saridakis, Valèrie Issarny, and Christophe Bidan. Customized Remote Execution of Web Agents. In *Thirty-First Hawaii International Conference on System Sciences (HICSS-31)*, pages 614–620, January 1998.

[40] Steve Vinoski. Where is Middleware? *IEEE Internet Computing*, 6(2):83–85, 2002.

[41] Apostolos Zarras and Valèrie Issarny. A Framework for Systematic Synthesis of Transactional Middleware. In *Middleware'98*, pages 257–272, The Lake District, England, September 1998.

[42] Apostolos Zarras and Valèrie Issarny. Imposing Transactional Properties on Distributed Software Architecture. In *Eighth ACM SIGOPS European Workshop - Suuport for Composing Distributed Applications*, pages 25–32, Sintra, Portugal, September 1998.

[43] Apostolos Zarras and Valérie Issarny. Assessing Software Reliability at the Architectural Level. In *Fourth International Software Architecture Workshop (ISAW-4)*, Limerick, Ireland, June 2000.