Process^{NFL}: A Language for Describing Non-Functional Properties

Nelson S. Rosa and Paulo R. F. Cunha
Centro de Informática
Universidade Federal de Pernambuco
Av. Prof. Luiz Freire, s/n
50732-970 Recife, Pernambuco - Brazil
nsr@cin.ufpe.br, prfc@cin.ufpe.br

George R. R. Justo
Centre for Parallel Computing
University of Westminster
115 New Cavendish Street,
London, UK - W1M 8JS
justog@cpc.wmin.ac.uk

Abstract

Non-functional requirements (NFRs) are rarely taken in account in most software development. 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 scenario, we concentrate on defining a language, namely Process^{NFL}*, that expresses NFRs during the software development. This language has been designed to consider specific characteristics of NFRs like their correlations and conflicts. In order to illustrate our language, we describe NFRs of software radios and the NFRs performance and security, which are usually present in the software requirements specifications.*

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 exhibited [4]. Functional requirements have typically localised effects, i.e., they affect only the part of software addressing the functionality defined by the requirement. On the other hand, NFRs specify global constraints that must be satisfied by the software, e.g., *performance*, *fault-tolerance*, *availability* and *security*.

During the software development, 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 the same way as the functional ones. They are usually satisfied in a certain degree, or *satisficed* [17], as a consequence of design decisions taken to implement the software's functionality.

NFRs are rarely considered when software systems are built, especially in the early stages of the software development. 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*" and "the component is *secure*" are common descriptions of NFRs; NFRs 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 whether the final product satisfies or not a specific non-functional property, 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 *performance*; 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 [5, 17, 22]. Firstly, there is an increasing demand for fault-tolerant, multimedia and real-time applications, in which NFRs play a critical role and their satisfaction are mandatory. Secondly, as a kind of requirement, it is natural their integration into the software development. Thirdly, interactions among functional and non-functional requirements are so strong in most cases that NFRs cannot 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 [22].

In order to address the problem of explicitly treating NFRs, two approaches have been traditionally adopted:

[1] Also referred to quality-attributes requirements [2], non-business requirements, goals [17], softgoals [18], QoS parameters [1], QoS requirements [12, 14], software quality factors [10], *ilities*[8] and afunctional qualities [6].

0-7695-1435-9/02 $17.00 (c) 2002 IEEE

process-oriented and product-oriented [17]. In the first approach, NFRs are viewed as effective elements in the software development 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 in which they are explicitly stated. In this approach, NFRs are measured and used to compare quality attributes of the software.

Most approaches proposed are product-oriented and concentrate on defining notations that can be used to express NFRs of the final product. Notations based on logic systems such as first-order logic [13], temporal logic [25] and predicate logic [23] express NFRs as predicates. A further strategy for describing NFRs is proposed by [9], in which a notation called NoFun is specially designed for this purpose. Other approaches are defined in Aspect-oriented languages [14], Pragma language [7] and a more formal one based on Z [21]. In relation to the process-oriented approaches, the NFR Framework has given a significant contribution [5]. It adopts a graphical notation for representing NFRs, their decompositions and correlations.

General issues, however, must be observed in the mentioned proposals. Firstly, most approaches act on an isolated way, e.g., notations for describing non-functional properties are designed, but they are not integrated with the functional requirements. Secondly, in a few cases in which the integration is proposed, it is accomplished in a very weak way, in the sense that the integration happens only at the end of the software development. Thirdly, notations based on logic systems demand a lot of effort by developers who have not a strong logic background. Finally, there is a unique process-oriented approach, but its graphical notation is not suitable to be integrated with notations used to describe functional requirements.

On this scenario, we present the language Process^{NFL} for describing non-functional properties. Unlike most mentioned notations, this language follows the process-oriented approach as strategy for treating with NFRs. Process^{NFL} has been designed with special skills in order to support particular characteristics of NFRs such as their strong correlations, often conflicts and non-direct implementation nature. Additionally, Process^{NFL} has ability for explicitly modelling the relationships between NFRs and the implementation elements that affect them. In order to carry out this task, the language has been defined around basic abstractions that represent our comprehension of NFRs and implementation elements, together with their relationships.

This paper is organised as following: Section 2 presents our view on how to reason about NFRs. Section 3 presents Process^{NFL} in details. Following section, Section 4, presents the Process^{NFL} description of *performance* and *security* and non-functional properties of software radios. Finally, last section presents the conclusions and some directions for future work.

2 Basic Principles

An initial step towards the definition of a notation for describing non-functional properties is to explain how to reason about them. Unlike functional requirements, which are usually "realised" through algorithms and data structures, non-functional ones lack of elements that can represent, implement and structure them. Therefore, our first step is the definition of abstractions, namely NF-Attribute, NF-Property and NF-Action, used to model non-functional issues. Additionally, as important as understanding the role of individual abstractions for representing non-functional properties, a key point to be taken in account is the relationship between them. As mentioned before, non-functional properties are usually in conflict and their correlations are particular characteristics to be considered in the modelling.

Next subsections present details of the mentioned non-functional abstractions and how they are related.

2.1 Non-Functional Attributes

Non-functional attributes (NF-Attributes) model non-functional characteristics that can be precisely measured like *performance*; non-functional features that cannot be quantified, but may be defined as required in the final product in a certain level like *security* and *availability;* and any non-functional aspect that must be simply present (without measure or level), like the transaction properties *atomicity*, *consistency*, *isolation* and *durability*. According to this distinctive nature of NF-Attributes, we have categorised them into ones precisely measured (Class 1), ones stated through levels (Class 2) and ones just present (Class 3) in the final product.

Another key characteristic of NF-Attributes is the possibility of decomposing them. A 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 primitives NF-Attributes. For a **composite** NF-Attribute, its primitive components participate in three different ways in order to compose 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.

0-7695-1435-9/02 $17.00 (c) 2002 IEEE
Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS-35'02)
0-7695-1435-9/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

The degree of decomposition of a 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 are a great number of NF-Attributes and their enormous variety, we have decided to focus on those related to runtime issues. For example, *performance*, *availability* and *security* are mainly related to a running software system, rather than issues of its development (non-runtime quality attributes [6]) such as *reusability*, *testability* and *modifiability*. The decision of selecting runtime properties has been motivated by some facts: most of them are included in the software requirement specification [11]; they are more visible to user's application instead of developers; their satisfaction have been increasing in most WEB applications; they are critical factors for the proper functioning of real-time and safety-critical systems; and they are directly affected by functional requirements.

2.2 Non-Functional Actions

A Non-Functional Action (NF-Action) models either any software aspect or any hardware characteristic that affect the NF-Attributes introduced in the previous section. Software aspects mean design decisions, algorithms, data structures and so on. Hardware characteristics concern computer resources available for running the software system. For example, the NF-Attribute *performance* is decomposed into two additional more primitive NF-Attributes, namely *space_performance* and *time_performance*. An algorithm (software aspect) used to compress data has a direct influence on *performance*, meanwhile the size of main memory and secondary storage capacity (hardware characteristics) are also important factors to be considered to achieve a *good performance*.

An important issue to be taken in account in the previous definition of NF-Action is the meaning of the statement "a NF-Action affects a NF-Attribute". A more precise consideration of "affects" reveals that it refers to both "realise" or have an "effect over" a non-functional aspect. By realising, the NF-Action acts in order to implement/operationalise the NF-Attribute. In relation to "effect over", it refers to have an influence over, i.e., a NF-Action whose effect over any NF-Attribute cannot be neglected.

The effect of a NF-Action over NF-Attributes is either *against* or *in favour* of them. For instance, *good performance* is not directly implemented, but there are NF-Actions that affect (or have an effect over) 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 about NF-Actions is the notion of "correlation". Correlation refers to the fact that a NF-Action implementing or affecting a NF-Attribute may also have an effect over other NF-Attributes. For example, the NF-Action *encryption algorithm* implements elements of the NF-Attribute *security*. Furthermore, this NF-Action also interferes in the NF-Attribute *performance*, as it is necessary to spend time with the execution of the encryption algorithm. According to our approach, the NF-Attributes *performance* and *security* are correlated.

Finally, as mentioned in Section 1, it is very often conflicts between non-functional properties. Conflicts appear as the result of correlations between NF-Attributes. For example, if a NF-Action has an *in favour* effect over *performance*, but it also has an *against* effect over another NF-Attribute, may it be implemented or not? The previous example presents a typical conflict of the NF-Attributes *performance* and *security*. We treat this kind of situation by allowing the assignment of priorities to NF-Attributes. The assignment of priorities is shown in the next section.

2.3 Non-Functional Properties

A Non-Functional Property (NF-Property) models constraints over NF-Attributes. Constraints impose conditions on what is required in the implementation of the software, in relation to a certain NF-Attribute. In practical terms, the constraint defines a subset of NF-Actions that must be actually implemented in order to satisfy the NF-Property. For example, the NF-Property *good performance* expresses a constraint over the NF-Attribute *performance* ("be good"). In this particular case, only NF-Actions that contribute to achieve a *good performance* must be taken in account.

The constraint is defined in terms of "levels", depending on the class of the NF-Attribute. NF-Properties defined over NF-Attributes of Class 1 and Class 2 (see Section 2.1) are expressed in terms of the following level of constraints:

- a "strong constraint" defines that every NF-Action that implements a NF-Attribute and every NF-Action that affects *in favour* of the NF-Attribute must be considered and none NF-Action that affects *against* the NF-Attribute must be realised;

- a "medium constraint" states that every NF-Action that implements the NF-Attribute and every NF-Action that affects *in favour* of the NF-Attribute also must be considered; and

IEEE COMPUTER SOCIETY

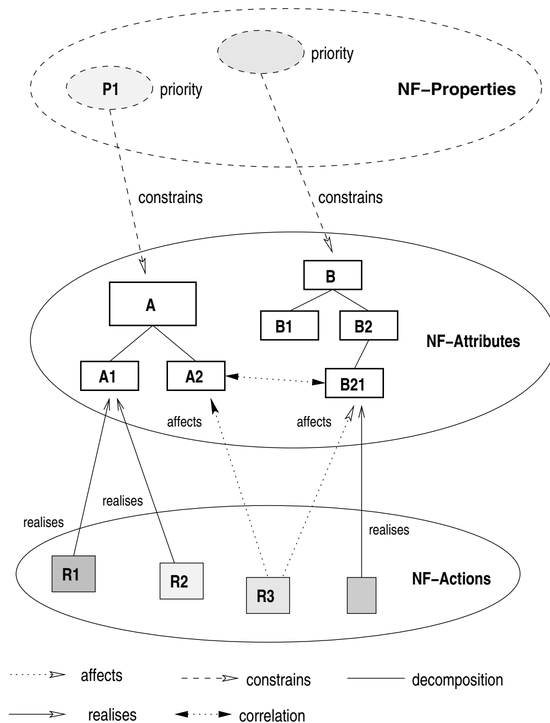

Figure 1. Non-Functional Abstractions

- a "low constraint" defines that every NF-Action that implements the NF-Attribute must be realised.

For NF-Attributes that belong to Class 3, the NF-Property definition consists simply of defining the presence or not of the NF-Attribute. For example, the NF-Attribute *atomicity* (Class 3) is not defined in a certain level, but the necessity of the *atomicity* is the constraint itself.

As mentioned in Section 2.2, conflicts are treated with the assignment of priorities to the NF-Attributes in the definition of NF-Properties. Three level of priorities are considered, namely high, medium and low. These priorities are defined in the NF-Property, together the constraints mentioned previously. For example, let the NF-Property *good_performance_secure* that constrains the NF-Attributes *performance* and *security*, it is also necessary to assign priorities in order to decide which NF-Actions must be considered to achieve the *good_performance_secure*. This is necessary, as *performance* and *security* are correlated and have a conflict.

Figure 1 shows a generic example of relationships between NF-Attributes, NF-Actions and NF-Properties. The NF-Property P1 constrains the NF-Attribute A, which is decomposed into the NF-Attributes A1 and A2. Three NF-Actions are related to these NF-Attributes, R1, R2 and R3. R1 and R2 realise the NF-Attribute A1, while R3 one only affects A2 and B21. According to our definition of correlation, the NF-Attributes A2 and B21 are correlated. Additionally, it is obligatory the implementation of the NF-Actions R1 and R2, while R3 is realised according to the constraint imposed on A and its priority.

3 Process^{NFL} Language

As stated previously, our approach adopts the strategy in which non-functional properties are treated during different steps of the software development. The traditional approach of defining a language to be used only when the software is already built, product-oriented approach, is obviously not useful in the software development. Product-oriented languages are suitable for precisely describing NF-Property, e.g., typical descriptions of the NF-Attribute *performance* state that a number such as 10 transactions/second represents it. However, in the initial steps of the development, this precise number is not available, because the software is not running yet. At this stage, what is necessary is to define which NF-Actions must be realised in order to achieve a certain NF-Property.

Some design principles have been followed in order to define Process^{NFL}:

- **Declarative Language**: Process^{NFL} is a declarative language able to describe non-functional properties at different levels of abstractions;

- **Usability**: the language has a simple structure made up of templates, which are used to describe the non-functional abstractions NF-Attribute, NF-Action and NF-Property. These generic templates enable us to describe runtime non-functional properties in a similar way, despite their distinctive nature;

- **First-class entities**: NF-Attributes, NF-Actions and NF-Properties are considered first-class entities in Process^{NFL};

- **Integration with functional requirements**: the integration with the functional part is explicitly placed in the NF-Action template, which serves for treating implementation issues. This template may be used both to explore functional characteristics of the software that affect a NF-Attribute and specific implementation elements, which are placed in the implementation for achieving a non-functional characteristic; and

- **NF-Actions like components**: following the idea of effectively integrating non-functional properties into the software development, NF-Actions may be thought as software architecture components [19, 24].

Next subsections present the Process^{NFL}'s templates.

0-7695-1435-9/02 $17.00 (c) 2002 IEEE
Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS-35'02)
0-7695-1435-9/02 $17.00 © 2002 IEEE

3.1 NF-Attribute Template

The non-functional abstraction NF-Attribute is described through two clauses grouped in a template as defined in the following:

attribute $attributeId_1$ **extends** $attributeId_2$;
{
primitives $primitiveAttributes$;
contribution $kindOfContribution$;
}

The NF-Attribute $attributeId_1$ extends the NF-Attribute $attributeId_2$. Process[NFL] defines a generic NF-Attribute, namely NFR, which is extended by all other NF-Attributes. The notion of extending is defined in order to enable us to create an hierarchy of NF-Attributes. The clause **primitives** contains the set of primitive attributes that contributes to $attributeId_1$, e.g., the NF-Attribute *performance* is traditionally decomposed into the primitive NF-Attributes *time_performance* and *space_performance*. A **simple** NF-Attribute (see Section 2.1) has the set of $primitiveAttributes$ empty (represented by none in Process[NFL]), while a **composite** NF-Attribute is decomposed into one or more primitive attributes.

The contribution ($kindOfContribution$) of primitive attributes to $attributeId_1$ is expressed through the clause **contribution**. Four different kind of contributions are considered in Process[NFL]:

- all: $attributeId_1$ is completely defined by composing all its primitive attributes;
- one+: $attributeId_1$ may be completely defined by one or more of its primitive attributes;
- oneX: $attributeId_1$ is completely defined by exactly one (any) of its attributes; and
- none: $attributeId_1$ is already completely defined, i.e., it is a **simple** NF-Attribute.

From our approach on how to describe a NF-Attribute, a basic rule is implicit: two **composite** NF-Attributes have not primitive NF-Attributes in common. It means that a primitive attribute only participates in the definition of a NF-Attribute.

3.2 NF-Action Template

A NF-Action (see Section 2.2) is defined in Process[NFL] through the template showing in the following

action $actionId$;
{
affected $setOfAffectedAttributes$;
implemented $setOfImplementedAttributes$;
effect
$attributeId_1$ [kindOfEffect];
...
$attributeId_n$ [kindOfEffect];
}

The clause **affected** defines which NF-Attributes ($setOfAffectedAttributes$) are affected by the implementation (or realisation) of $actionId$. The second clause, **implemented**, contains the set of NF-Attributes ($setOfImplementedAttributes$) directly implemented by $actionId$. Finally, the last clause, **effect**, defines how the implementation of $actionId$ affects individual NF-Attributes belonging to $setOfAffectedAttributes$. For each affected NF-Attribute is defined the kind of effect (kindOfEffect): +1, +2, +3 and -1. The first three degrees are used to model an *in favour* effect, in which +3 is stronger than +2 that is stronger than +1. The degree -1 is only used to define an *against* effect of $actionId$ over the NF-Attribute.

Three basic facts emerges from our treatment of NF-Actions:

- A NF-Action either implements or affects a NF-Attribute: if an $attributeId_i$ is implemented by $actionId$, the attribute is not affected by the same $actionId$. In a similar way, if an $attributeId_j$ is affected by $actionId$, it cannot be implemented by the same $actionId$;
- A NF-Action either implements or affects at least one NF-Attribute: $actionId$ is defined in Process[NFL] only if it affects a NF-attribute; and
- A NF-Action only affects or implements **simple** NF-Attributes: **composite** attributes are not directly affected by $actionId$.

3.3 NF-Property Template

The last template describes the abstraction NF-Property (see Section 2.3). This template put together elements defined in the **attribute** and **action** templates. It is defined as depicted in the following

property *propertyId*;
{
constraints
attributeId$_1$ [kindOfConstraint];
...
attributeId$_n$ [kindOfConstraint];
priorities

```
        attributeId_1 [ kindOfPriority ];
        ...
        attributeId_n [ kindOfPriority ];
    }
```

A $propertyId$ is defined by setting constraints (clause **constraints**) over individual NF-attributes $attributeId_1$, ..., $attributeId_n$. Process^{NFL} defines four different kind of constraints (kindOfConstraint) imposed to individual NF-Attributes: weak, light, strong and present (see Section 2.3). Additionally, the template allows the assignment of priorities to each NF-Attribute (clause **priorities**). Three different levels of priority (kindOfPriority) are defined in the language: low, medium and high. As the names suggest, high priority is higher than medium and medium is higher than low.

A typical Process^{NFL} specification contains the definition of one or more NF-Attributes, one or more NF-Actions and one NF-Property:

```
attribute attributeId_1 extends attributeId_2;
  ...
attribute attributeId_3 extends attributeId_4;
  ...
action actionId_1;
  ...
action actionId_2;
  ...
action actionId_3;
  ...
property propertyId;
  ...
```

4 Examples

This section presents how Process^{NFL} is used to describe non-functional properties. An initial example expresses the two well known non-functional properties *security* and *performance* through the definition of the NF-Property *good_performance_secure*. The second example concentrates on the description of non-functional properties of software radios [3, 15].

4.1 *Performance* and *Security*

This example was firstly specified through the graphical notation of the NFR Framework [5]. According to our approach, *performance* is a **composite** NF-Attribute defined in terms of two primitives non-functional attributes: *space_performance* and *time_performance*. These NF-Attributes have a oneX contribution, expressing that *performance* may be defined either considering NF-Actions related to *space_performance* or *time_performance*. Both *space_performance* and *time_performance* are **simple** NF-Attributes, as they have not been decomposed into more primitive NF-Attributes.

```
attribute performance extends NFR;
  {
    primitives space_performance,time_performance;
    contribution oneX;
  }

attribute space_performance extends performance;
  {
    primitives none;
    contribution none;
  }

attribute time_performance extends performance;
  {
    primitives none;
    contribution none;
  }
```

In terms of *security*, it is made up of three primitive NF-Attributes: *integrity*, *confidentiality* and *availability*. All of them are necessary in the decomposition of *security* (contribution all). In a similar way as *performance*, *security* also has only one level of decomposition, as its primitive attributes are **simple** NF-Attributes.

```
attribute security extends NFR;
  {
    primitives integrity,confidentiality,availability;
    contribution all;
  }

attribute integrity extends security;
  {
    primitives none;
    contribution none;
  }

attribute confidentiality extends security;
  {
    primitives none;
    contribution none;
  }

attribute availability extends security;
  {
    primitives none;
    contribution none;
  }
```


Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS-35'02)
0-7695-1435-9/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

There are two different sets of NF-Actions, one related to *performance* issues and others referring to *security*. As *performance* is a NF-Attribute not directly implemented, NF-Actions only affect it. These NF-Actions are *useIndexing* and *useCompressedFormat*. *UseCompressedFormat* affects the NF-Attributes *space_performance* (*in favour*, +3) and *time_performance* (*against*, -1). The *against* effect of *useCompressedFormat* over *time_performance* is consequence of the time spent to compress data. In relation to *useIndexing*, it has an *in favour* effect (+3) over *time_performance*.

It is worth noting that *time_performance* and *space_performance* are correlated, as they have the NF-Action *useCompressedFormat* affecting both of them. Additionally, there is also a conflict between them, because *useCompressedFormat* has an opposite effect over *space_performance* (*in favour*, +3) in relation to *time_performance* (*against*, -1).

```
action useCompressedFormat;
  {
     affected space_performance,time_performance;
     implemented none;
     effect
          space_performance [+3];
          time_performance [-1];
  }

action useIndexing;
  {
     affected space_performance;
     implemented none;
     effect
          time_performance [+3];
  }
```

Security has a different nature from *performance*, as the NF-Actions related to it have the role of directly implementing *security* issues. NF-Actions that may be implemented to obtain a *secure* system are all of them related to the NF-Attribute *confidentiality*: *validateAccessAgainstEligibilityRules*, *identifyUser*, *usePIN*, *compareSignature* and *requireAdditionalId*.

```
action validateAccessAgainstEligibilityRules;
  {
     affected time_performance;
     implemented confidentiality;
     effect time_performance [-1];
  }

action identifyUser;
  {
     affected none;
     implemented confidentiality;
     effect none;
  }

action usePIN;
  {
     affected none;
     implemented confidentiality;
     effect none;
  }

action compareSignature;
  {
     affected none;
     implemented confidentiality;
     effect none;
  }

action requireAdditionalId;
  {
     affected none;
     implemented confidentiality;
     effect none;
  }
```

It is worth observing that the NF-Attributes *confidentiality* and *response_time* are also correlated, as the NF-Action *validateAccessAgainstEligibilityRules* affects both of them. In a similar way, there is also a conflict between them. The conflict appears because *validateAccessAgainstEligibilityRules* implements *confidentiality*, while has an *against* (-1) effect over *response_time*.

Finally, the NF-Property *good_performance_secure*, as mentioned before, is a constraint over *performance* and *security*. We assume that the constraint imposed to *security* is stronger than ones imposed to *performance*, as the kind of constraint (*kindOfConstraint*) is strong and medium, respectively. Additionally, the priority assigned to *security* (high) is higher than one associated to *performance* (low). It means that NF-Actions assigned to *security* are implemented preferable to ones related to *performance*.

```
property good_performance_secure;
  {
     constraints
          performance [ medium ];
          security [ strong ];
     priorities
          performance [ low ];
          security [ high ];
  }
```

4.2 Non-Functional Properties of Software Radios

In this example, we illustrate how Process[NFL] may used to describe non-functional properties of software radios. Software radio [3, 15] is a term used to refer to the convergence of today's digital radio and software technologies. The key point of this convergence comes from the fact that communication technologies continue its rapid transition from analogue to digital. As a consequence of this transition, more functions of contemporary radio systems are implemented in software - leading toward the software radio (SR)[2] . As functions are defined in software (e.g., in JAVA), issues of the "traditional" software development like the treatment of non-functional properties also must be taken in account in the context of software radios.

Non-functional properties of software radios are related to *processing*, *memory capacity*, *real-time performance* and *power consumption.* In particular, the necessity for *real-time performance* ($performance$) is a critical property for the proper functioning of software radios.

Performance is defined through the demand of three main resources, namely I/O bandwidth, memory and processing capacity [16]. Hence, according to our approach, the NF-Attribute $performance$ is made up of more primitive NF-Attributes, namely $processing_capacity,$ $IO_bandwidth_capacity$ and $memory_capapcity$. All of them are necessary in the definition of $performance$

attribute $performance$ extends NFR;
{
primitives
$processing_capacity$,
$io_bandwidth_capacity$,
$memory_capapcity$;
contribution all;
}

For simplicity and lack of space, we concentrate on the NF-Attribute $processing_capapcity$. It is defined in terms of the IF[3] processing ($if_processing_capacity$), baseband processing ($baseband_processing_capacity$), bitstream processing ($bistream_processing_capacity$), source processing ($source_processing_capacity$) and overhead processing ($overhead_processing_capacity$)

attribute $processing_capapcity$ extends $performance$;
{
primitives
$if_processing_capacity$,
$baseband_processing_capacity$,
$bitstream_processing_capacity$,
$source_processing_capacity$,
$overhead_processing_capacity$;
contribution all;
}

[2] The terms adaptive terminal [20], software defined radio (SDR), reconfigurable radio systems and networks and cognitive radios are used to refer to software radio.

[3] Intermediate Frequency.

First primitive NF-Attribute of $processing_capapcity$, $if_processing_capacity$, is defined in terms of the bandwidth of the accessed service band ($bw_accessed$), per-point complexity of the service band isolation filter ($g_isolation$) and the complexity of subscriber channel-isolation filters ($g_subscriber$)

attribute $if_processing_capacity$ extends
$processing_capacity$;
{
primitives $bw_accessed$,$g_isolation$,$g_subscriber$;
contribution all;
}

Second primitive NF-Attribute, $baseband_processing_capacity,$ is defined in terms of the bandwidth of a single channel ($w_channel$), complexity of modulation processing and filtering (g_profil), processing of demodulation ($g_modulation$)

attribute $baseband_processing_capacity$ extends
$processing_capacity$;
{
primitives $w_channel, g_profil, g_modulation$;
contribution all;
}

Third NF-Attribute, $bitstream_processing_capacity,$ is defined through the composition of the data rate of the bitstream ($b_datarate$), code rate ($code_rate$) and the per-point complexity of bitstream processing ($g_bitstream$)

attribute $bitstream_processing_capacity$ extends
$processing_capacity$;
{
primitives $b_datarate, code_rate, g_bitstream$;
contribution all;
}

Finally, last two primitive NF-Attributes, $source_processing_capacity$ and $overhead_processing_capacity$, are simple NF-Attributes.

Using the decomposition and definition of the NF-Attribute $performance$ proposed above, the NF-Property

0-7695-1435-9/02 $17.00 (c) 2002 IEEE

$real_time_performance$ is defined as follows

property $real_time_performance$;
constraints
$io_bandwidth_capacity$ [strong];
$memory_capacity$ [strong];
$processing_capacity$ [strong];
priorities
$io_bandwidth_capacity$ [low];
$memory_capacity$ [medium];
$processing_capacity$ [high];
}

According to this description, all NF-Attributes impose strong constraints over the NF-Attribute $performance$, while $processing_capacity$ has the highest priority. It means that NF-Actions that implement of affect the NF-Attribute $processing_capacity$ are considered priorities to be implemented.

5 Conclusion and Future Work

This paper has presented the Process[NFL] language for describing non-functional properties during the software development. Non-functional properties are expressed through three abstractions, NF-Attributes, NF-Properties and NF-Actions. These abstractions are described through templates in Process[NFL]. In addition to concentrate on the description of individual abstractions, the language also makes explicit the relationships between them. Essentially, Process[NFL] treats non-functional properties defining how to express them at a high abstract level (NF-Attributes), how to express constraints over them (NF-Properties) and how design decisions/hardware aspects (NF-Actions) must be considered in order to achieve the constraint imposed over NF-Attributes.

Benefits of our approach comes from the simplicity of the language, the possibility for describing non-functional properties in different level of abstractions (through their decomposition) and the way how to face the complexity of treating non-functional properties, i.e., through a set of well defined abstractions. Additionally, Process[NFL] can be adopted to communicate non-functional properties in a more precise way, while also may be used to create a catalogue of non-functional properties.

Obviously, this proposal does not resolve all the problems related to the description of non-functional properties. This is a very complex task that comprises essentially a stronger formalisation task, a better understanding of specific properties and further studies about the distinctive nature of the non-functional properties themselves. However, Process[NFL] is an effective step towards the explicit treatment of this kind of property, as the language enable us to understand in a more clear way how to reason about them. Additionally, the proposed language is easier to be used if compared with logical notations and enable a more concrete view of non-functional properties and their correlations.

As the treatment of non-functional properties is only in the beginning, additional points have to be investigated. Firstly, a formal semantics has to be defined for the language, which enable us to verify properties of the non-functional properties, e.g., absence of conflicts. Secondly, the notation have to be used to describe further properties. Finally, the language must actually be integrated with current languages and methodologies, e.g., the implementation of a NF-Action may be written in a common programming language.

References

[1] L. Blair and G. Blair. Composition in Multi-Paradigm Specification Techniques. In *FMOODS'99*, Florence, Italy, Feb. 1999.

[2] B. Boehm and H. In. Identifying Quality Requirements Conflicts. *IEEE Software*, 13(2):25–35, Mar. 1996.

[3] E. Buracchini. The Software Radio Concept. *IEEE Communication Magazine*, 38(9):138–143, Sept. 2000.

[4] L. Chung. Representation and Utilization of Non-functional Requirements for Information System Design. In *3rd International Conference on Advanced Information System Engineering - CAiSE'91*, Trondheim, Norway, May 1991.

[5] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-functional Requirements in Software Engineering*. Kluwer Academic Publishers, 1999.

[6] P. C. Clements. Coming Attractions in Software Architecture. Technical Report CMU/SEI-96-TR-008, Software Engineering Institute, Carnegie Mellon University, Jan. 1996.

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

[8] R. E. Filman. Injecting Ilities. In *Aspect-Oriented Programming Workshop, ICSE'98*, Kyoton, Japan, Apr. 1998.

[9] X. Franch. The Convenience for a Notation to Express Non-functional Characteristics of Software Components. In *Foundations of Component-based Systems Workshop (FoCBS)*, pages 101–109, Zurich, Switzerland, Sept. 1997.

[10] A. K. Ghose. Managing Requirements Evolution: Formal Support for Functional and Non-functional Requirements. In *International Workshop on the Principles of Software Evolution (IWPSE'99)*, Fukuoka, Japan, July 1999.

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

[12] V. Issarny and C. Bidan. Aster: A Framework for Sound Customization of Distributed Runtime Systems. In *16th International Conference on Distributed Computing Systems*, pages 586–593, Hong Kong, May 1996.

[13] V. Issarny, C. Bidan, and T. Saridakis. Achieving Middleware Customization in a Configuration-based Development Environment: Experience with the Aster Prototype. In *4th International Conference on Configurable Distributed Systems*, pages 207–214, Annapolis, Maryland, USA, 1998.

IEEE COMPUTER SOCIETY

[14] J. P. Loyall, D. E. Bakken, R. E. Schants, J. A. Zinky, D. A. Karr, R. Vanegas, and K. 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.

[15] J. Mitola. The Software Radio Architecture. *IEEE Communications Magazine*, 33(5):26–38, May 1995.

[16] J. Mitola. *Software Radio Architecture - Object-Oriented Approaches to Wireless Systems Engineering*. John Wiley & Sons, Inc., 2000.

[17] J. Mylopoulos, L. Chung, and B. Nixon. Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. *IEEE Transaction of Software Engineering*, 18(6):483–497, June 1992.

[18] J. Mylopoulos, L. Chung, and E. Yu. From Object-oriented to Goal-oriented Requirement Analysis. *Communications of the ACM*, 42(1):31–37, Jan. 1999.

[19] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *Software Engineering Notes*, 17(4):40–52, Oct. 1992.

[20] B. Robinson. Software Radio: The Standards Perspective. In *CEC Software Radio Workshop*, Brussels, May 1997.

[21] N. S. Rosa, G. R. R. Justo, and P. R. F. Cunha. Incorporating Non-Functional Requirements into Software Architecture. In *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.

[22] N. S. Rosa, G. R. R. Justo, and P. R. F. Cunha. A Framework for Building Non-Functional Software Architectures. In *16th ACM Symposium on Applied Computing*, pages 141–147, Las Vegas, USA, Mar. 2001.

[23] T. Saridakis and V. Issarny. Fault Tolerant Software Architectures. Technical Report 3350, INRIA, 1998.

[24] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

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

0-7695-1435-9/02 $17.00 (c) 2002 IEEE

Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS-35'02)
0-7695-1435-9/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY