A Framework for Building Non-Functional Software Architectures

Nelson S. Rosa
Universidade Federal de Pernambuco
Centro de Informática
Av. Prof. Luiz Freire, s/n
Recife, Pernambuco, Brazil
nsr@cin.ufpe.br

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

Paulo R. F. Cunha
Universidade Federal de Pernambuco
Centro de Informática
Av. Prof. Luiz Freire, s/n
Recife, Pernambuco, Brazil
prfc@cin.ufpe.br

ABSTRACT
Non-functional requirements (NFRs) are rarely taken into account in most software development processes. There exist reasons that can help us to understand why these requirements are not explicitly dealt with: their complexity, their usually informal statement, their high abstraction level, as well as the rare support of languages, methodologies and tools for them. In this paper, we introduce a framework for explicitly dealing with NFRs in the software development process. This framework addresses the description, integration with functional requirements, refinement and mapping of NFRs into actual implementation elements.

1. INTRODUCTION
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 safe” 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 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 competing 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 formally and explicitly with NFRs is apparent [12, 2]. Firstly, there is an increasing demand for fault-tolerant, multimedia and real-time applications, in which NFRs play a critical role and their satisfaction is mandatory. Secondly, as a kind of requirement, it seems natural to integrate it into the software development. Thirdly, a formal approach makes possible to verify properties of NFRs, which is a basic step (for some of NFRs) towards an automatic treatment of NFRs. Fourthly, 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.

Two approaches have been traditionally adopted in order to address the problem of explicitly dealing with NFRs: process-oriented and product-oriented [12]. 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. NFRs are measured and used to compare quality attributes of the software. Most approaches previously 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 [5], temporal logic [19] and predicate logic [17] express NFRs as predicates. A further strategy for describing NFRs is proposed by [4], in which a notation called NoFun was specially designed for this purpose. Other approaches are defined in Aspect-oriented languages [8], Pragma language [3] and a more formal one based on Z [15]. In relation to the
Considering these proposals, general problems can be identified. Firstly, most approaches act on an isolated way, e.g., notations are defined, but they are not integrated with other steps of the software development. Secondly, the notation based on logic systems makes difficult their practical use by developers who have not a strong logic background. Thirdly, 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 development. Finally, there exist a unique process-oriented approach, but its graphical notation makes difficult its integration with possible notations used to describe functional requirements.

In this paper, we propose the framework Parmenides for dealing with NFRs in the steps of the software development process. In relation to the previous proposals, it is a hybrid approach in which process- and product-oriented strategies are adopted together. The framework consists of a process-oriented language for describing NFRs (at the begin of the development), a guidelines on how to incorporate NFRs into the software architecture (defining what we designate non-functional software architectures), a set of rules for refinement of non-functional architectures, an strategy for mapping non-functional architectures into actual implementation elements and a product-oriented language suitable for describing NFRs at the final product.

This paper is organised as following: Section 2 introduces basic concepts of NFRs and the software architecture. Section 3 presents the Parmenides framework in a detailed manner. Finally, the last section presents an analysis of the framework, the conclusions and some directions for future work.

2. BACKGROUND

Basic concepts of software architectures play a main role in the proposed framework. Architectural elements such as component and connectors are the basic abstraction units to which NFRs will be assigned. Additionally, it is necessary a more complete description of NFRs. The following subsections present basic concepts of NFRs and software architecture.

2.1 Non-Functional Requirements

Functional requirements define what a software is expected to do, while non-functional requirements specify global constraints on how the software operates or how the functionality is exhibited [1, 2]. Functional requirements usually have localised effects, i.e., they affect only the part of the software addressing the functionality defined by the requirement. During the software development process, functional requirements are usually incorporated into the software artifacts step by step. At the end of the process, all functional requirements must have been implemented in such way that the software satisfies the requirements defined at the early stages. NFRs, however, have a global nature, which means that to satisfy a NFR may affect several design components. Thus, NFRs are not implemented in the same way as functional ones.

NFRs have a very distinctive nature, in which a wide variety of aspects such as modifiability and fault-tolerance are categorised as non-functional properties. The IEEE/ANSI 830-1993, IEEE Recommended Practice for Software Requirements Specifications defines thirteen non-functional requirements that must be included in the software requirements document: performance, interface, operational, resource, verification, acceptance, documentation, security, portability, quality, reliability, maintainability and safety. Kotonya [7] classifies these requirements into three main categories: Product requirements, Process requirements and External requirements. Product requirements specify the desired characteristics that a system or subsystem must possess. Process requirements put constraints on the development process of the system. External requirements are constraints applied to both the product and the process which are derived from the environment where the system is developed.

2.2 Software Architecture

The software architecture [14, 18] is the highest abstract description of a software design, which is defined at the initial stages of the software development. Software architectures are commonly described in terms of three basic abstractions: components, connectors and configurations. Components represent a wide range of different elements, from a single client to a database, and have an interface (made up of ports) used to communicate the component with the external environment. Connectors represent communication elements between components. The configuration describes how components and connectors are wired.

Essentially, the software architecture presents a description of the software where “computation” (included in components) and “communication” (modelled by connectors) are clearly separated. In order to describe software architectures, languages specially designed for this purpose, namely Architecture Description Languages (ADLs), replace box-and-arrows diagrams or natural languages. This class of languages has some key characteristics such as components and connectors as first-class elements, ability for expressing NFRs and the focus on the structure of the software architecture [10].

3. THE PARMENIDES FRAMEWORK

Parmenides is a framework that considers NFRs within the entire software development process. The framework defines precisely how NFRs are expressed and integrated into an architecture-based development. As depicted in Figure 1, Parmenides “acts” on five different steps: proposes a language for an initial description of the NFRs, namely Process-NFL, defines how NFRs can be assigned to architectural elements, defines rules to guide the refinement, proposes a mapping strategy and defines a second language, Product-NFL, used to define the NFRs of the final product.

1Parmenides (born c. 515) is one of the most important Presocratic philosophers.
3.1 Basic Principles

Prior to describe individual parts of the framework, it is necessary to define basic elements and ideas that make up the core of Parmenides.

The first decision taken in Parmenides was to define the set of NFRs considered. This decision is necessary, because there is both a great number and a enormous variety of NFRs. As we are interested in the development of dynamic systems [6], requirements related to runtime issues such as performance, security and availability play an important role in their construction. Hence, the framework deals with a subset of the Product-oriented requirements (see Section 2.1).

As the set of NFRs has been defined, an important decision is how to reason about NFRs in a software development view. In the framework, NFRs are fundamentally viewed as constraints on possible design decisions for implementing the software. NFRs put constraints on how the functional part will work. Hence, every decision taken for implementing a functionality of the software must respect the constraints imposed by the non-functional part of the requirements.

![Diagram of Parmenides Framework](image)

**Figure 1: Overview of Parmenides**

The next decision in the framework was to define the set of abstractions necessary for expressing this idea of constraint. Any non-functional information, including requirements and properties of the software, is modelled in the Parmenides by three abstractions: NF-Attribute, NF-Realisation and NF-Requirement. A NF-Attribute models both any non-functional characteristic of the software that can be precisely pointed out (performance) and any non-functional feature that cannot be quantified, but may be defined as present in the software in a certain level (security). NF-Requirements are constraints over the NF-Attributes, e.g., good performance is a constraint on the NF-Attribute performance. NF-Realisations model design decisions or actions which may be followed to achieve a certain NF-Requirement, i.e., NF-Realisations act as design constraints, if they are adopted the NF-Requirement can be achieved. In general terms, we can state that NF-Requirements imposed to NF-Attributes are achieved if certain NF-Realisations are chosen.

In relation to these abstractions and their relationships, there are some points to be considered:

- NF-Attributes and NF-Realisations may be decomposed into sub-attributes and sub-realisations, respectively. For example, the NF-Attribute performance may be decomposed into space and time NF-Attributes, while an authorising access (a NF-Realisation for “implementing” the NF-Attribute Secure) may be carried out comparing signature or using a PIN.

- NF-Attributes are always decomposed into the same elements, while NF-Realisations are particular to a specific application. For example, the NF-Attribute performance is always decomposed into space and time, but design decisions to obtain a good performance must be different if it is assigned to distinct architectural elements (see Section 2.2);

- The idea of correlation and contribution among NFRs, as proposed in the NFR Framework [2], is adopted here as well. It means that NF-Attributes and NF-Realisations are correlated and contributes in different forms to others. A “positive” correlation means that the attribute acts “in favor of” another one, while a “negative” correlation has an opposite effect. In terms of contribution, if an NF-Attribute is decomposed into other attributes, all its sub-attributes are necessary to achieve it (“and” contribution) or at least one of the sub-attributes is necessary (“or” contribution);

- NF-Requirements are not directly realised, but it is possible to define a set of NF-Realisations that can help to achieve them. For example, one does not directly implement a set of NF-Realisations that can be favourable to obtain a good performance;

All the Parmenides elements have been formally defined [16]. For instance, the NF-Attribute was defined as following:

**NF-Attribute** is a 3-tuple \((\text{Id}_{\text{Att}}, \Sigma_{\text{Att}}, \text{F}_{\text{Cont}})\), where \(\text{Id}_{\text{Att}}\) is the identification of the NF-Attribute, \(\Sigma_{\text{Att}}\) is the set of sub-attributes of the NF-Attribute, \(\Delta_{\text{Cont}}\) is the set of the kind of contribution of sub-attributes, \(\text{F}_{\text{Cont}} : 2^{\Sigma_{\text{Att}}} \rightarrow \Delta_{\text{Cont}}\) defines the kind of contribution of the sub-attributes to the NF-Attribute.

After the introduction of these basic concepts, next sections introduce details of each component of Parmenides.

3.2 Process-NFL Language

Parmenides adopts a strategy in which NFRs are treated during the entire software development process. 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 initial steps of the software development. Product-oriented languages are suitable for precisely describing NFRs, e.g., typical descriptions of performance state that it is represented by a number such as 10 transactions/second. However, in the initial steps of the development, this information has no meaning, because the software is not running yet. Hence, a process language has different skills than a product-oriented one.

Process-NFL provides elements for expressing and composing NFRs, defining their correlation and contributions, and keeping track of the design options that can be taken for achieving NFRs. In order to perform this task, Process-NFL contains templates for representing NF-Attributes, NF-Realisations and NF-Requirements. The template for expressing NF-Attributes is shown in Figure 2.

```
attribute attributeId;
import importedAttributes;
component listOfSubAttributes;
contribution [kindOfContribution ]
```

Figure 2: Process-NFL attribute template

The clause `import` lists the NF-Attributes (`importedAttributes`) necessary within the definition of the current one (`attributeId`). Imported attributes may compose the set of attributes included in the clause `components`. This clause contains the sub-attributes that make up the attribute being defined, e.g., performance is traditionally decomposed into time performance and space performance. Sub-attributes contribute (clause `contribution`) in different ways to achieve the attribute being defined. According to [2], sub-attributes typically have a “and” or “or” contribution to the attribute.

It is worth noting that the structure of this template has two major benefits: the clause `components` allows to create an hierarchy of NF-Attributes, as it permits the decomposition of attributes into sub-attributes; and NF-Attributes definitions can be reused through the clause `import`.

The second template defined in Process-NFL allows the definition of NF-Realisations. This template contains the same syntactical elements as the previous one. However, semantically, NF-Attributes and NF-Realisations are completely different. NF-Realisations models something that can be implemented, while the previous template models something very abstract. For instance, a typical manner to obtain the NF-Requirement good performance is to use the NF-Realisation that indexes a set of data (`useIndex`). `useIndex` makes the access to an information faster. NF-Realisations can be reused within the definition of another NF-Realisation, may be decomposed, have contributions similar to NF-Attributes and also may be correlated to others.

The third template describes NF-Requirements. This template put together `attribute` and `realisation` templates. As the attribute template, the description of NF-Requirement also has an `import` clause which made available the NF-Attributes and NF-Realisations necessary within the requirement definition. An additional clause, `constraints`, defines which NF-Realisations can be used to implement the NF-Attributes in order to achieve the constraints imposed by the NF-Requirement. The list of NF-Realisations can change depending on the NF-Requirement that have to be achieved. For example, NF-Realisations used to implement a high performance must be different of others for implementing a high performance and security level 2.

A typical Process-NFL specification contains the definition of many NF-Attributes, many NF-Realisations and one NF-Requirements.

### 3.3 Integration

In this part of Parmenides, a strategy of how to integrate functional and non-functional elements is defined. The functional part is described within the software architecture, while the non-functional one has been described by Process-NFL. Essentially, the framework defines that NF-Requirements have to be assigned to architectural elements, namely connectors, components, ports, interfaces and the entire configuration. In terms of development, it means that the functional part passes 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-Requirement. We name non-functional software architecture (NF-Architecture), a software architecture in which NF-Requirements are explicitly defined. Similarly, we designate NF-Components, NF-Connectors, NF-Interfaces and NF-Ports for defining non-functional components, connectors, interfaces and ports, respectively.

The Architecture is the coarse-grain non-functional element, while the port represents the fine-grain one. In this way, the constraint imposed to an Architecture is applied to their components and connectors. Similarly, constraints imposed to components and connectors are applied to their respective interfaces and ports, and constraints imposed to the interface is also applied to their ports. For example, the framework assumes that the security of the entire configuration depends on the security of its components and connectors, the security of a component depends on the security of its interface and the security of the interface depends on the security of its ports. In the integration process, it is worth noting that: NFRs are usually assigned to connectors, rather than other architectural elements; and the integration must occur before any refinement of the functional part, as the constraint imposed by the NFRs can affect the functional part.

The initial ideas of the integration strategy were introduced in [15]. However, the integration was defined within a formal framework for describing dynamic software architectures based on the Z notation [13].

### 3.4 Refinement Rules

After being described and integrated with software architecture elements, it is time for refining the non-functional software architecture. Refinement rules define how a concrete non-functional architecture is obtained from an abstract non-functional one (see Figure 1). Hence, it is necessary to place together the refinement of non-functional
and architectural elements, i.e., Component, Connector, Interface, NF-Attribute, NF-Realisation and NF-Requirement that are present at the non-functional software architecture.

Moriconi [11] defines that architectural elements (components, connectors and interfaces) are refined by replacing, decomposing, aggregating and removing them. On the other hand, the refinement of NFRs consists of decomposing NF-Attributes and NF-Realisations [2]. In Parmenides, these elements are placed together, following some basic constraints: NF-Attribute decompositions are performed before any other refinement; NF-Attribute decompositions are universal in the sense that their refinement is independent of the architectural element to which it is assigned; NF-Realisation refinement is dependent on the architectural element to which it is associated to; and a NF-Requirement is refined, as their NF-Attributes and NF-Realisations are refined.

Following these constraints, refinement rules are categorised into four different classes: decomposition, aggregation, elimination and replacement rules. These rules are informally presented in Figure 3 (black boxes represent the element being refined) and formally introduced in [16]. It is worth noting that for lack of space, the refinement rules presented refers only to Components. However, all of them have a version for the other architectural elements.

### 3.4.1 Decomposition Rules

Decomposition rules are applied for decomposing non-functional (NF-Attributes and NF-Realisations) and architectural elements.

**Rule 1** Decomposition of an attribute into sub-attributes: this rule decomposes the NF-AttributeA into NF-AttributeA1 and NF-AttributeA2. Rule 1 may be applied if no realisation has been defined to the NF-AttributeA.

**Rule 2** Decomposition of a component: this rule decomposes the ComponentA into ComponentA1 and ComponentA2. After the decomposition, both components have the same NF-Requirements as the original decomposed component (ComponentA).

**Rule 3** Decomposition of realisations into sub-realisations: this rule decomposes the NF-RealisationA into NF-RealisationA1 and NF-RealisationA2.

### 3.4.2 Aggregation Rules

Aggregation rules are only applied to architectural elements, as there is no meaning the aggregation of NF-Attributes and NF-Realisations.

**Rule 4** Aggregation of components: this rule aggregates ComponentA1 and ComponentA2 into ComponentA. It is worth noting that architectural elements are only aggregated if their NF-Requirements are compatible.

### 3.4.3 Elimination Rules

As aggregation rules, elimination ones are applied to architectural elements. NF-Attributes and NF-Realisations are not eliminated in a specification.

**Rule 5** Elimination of NF-Components: this rule eliminates the ComponentB.

### 3.4.4 Replacement Rules

Replacement rules are applied to substitute a component by another more refined. Basically, the new component must satisfy the constraints imposed by the original NF-Requirements.

**Rule 6** Replacement of components: the ComponentB is replaced by another component more refined (ComponentB). The NF-Requirement of the original NF-Component must be preserved.

### 3.5 Mapping Strategy

The mapping strategy defines how the concrete non-functional software architecture, obtained as the result of the refinement, may be implemented into actual implementation...
elements. It means that architectural elements (components and connectors) and NF-Realisations (defined within NF-Requirement) have to be mapped into execution elements. The scenario where the mapping is carried out contains COTS\(^3\) (Commercial-Of-The-Shelf), which are stored in a library of COTS, and implementation environments such as CORBA, EJB \([9]\) and CL Environment \([6]\).

Components and connectors are mapped into COTS and implementation environments, respectively. In relation to the NF-Realisations, three basic situations may be identified: the COTS itself already implements the necessary NF-Realisation, e.g., a component that demands good performance may be mapped into a fast COTS; the NF-Realisation of the connector is usually implemented by the implementation environment, e.g., CORBA already provides security and transactional services; and the NF-Realisation may be partially implemented by the COTS itself and by specialised COTS that implements the NF-Realisation, e.g., a COTS that implements an authorisation access.

### 3.6 Product-NFL Language

Product-NFL is a notation for expressing non-functional properties of the final product. When the software is already built, following the previous steps of the framework, it is possible to define non-functional characteristics of the software in a more precise way. A Product-oriented language must basically describe the same kind of information as a process language, but in a precise manner. In practical terms, while Process-NFL expresses design decisions that can be followed if a certain NF-Requirement have to be achieved, Product-NFL must express the practical result of the application of the constraints imposed by the NF-Requirements. It is worth noting that the word “requirement” is replaced by “property” in order to express the idea that the requirement has been concretised and it has become an actual property of the final product.

<table>
<thead>
<tr>
<th>attribute attributed;</th>
<th>attribute Performance;</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>import importedAttribute;</td>
<td>import none;</td>
</tr>
<tr>
<td>component importAttributes;</td>
<td>component time : (\text{Real}); space : (\text{Integer});</td>
</tr>
<tr>
<td>contribution ([\text{kindOfContribution}]);</td>
<td>contribution ([\text{of} ; \text{time}]);</td>
</tr>
<tr>
<td>constraints predicate;</td>
<td>constraints time [seconds] &gt; 0.0; space [Mbytes] &gt; 0.</td>
</tr>
</tbody>
</table>

**Figure 4: Product-NFL attribute template**

Product-NFL expresses non-functional properties through templates, like Process-NFL. The first template defines NF-Attributes (see Figure 4). The clauses of the description of NF-Attributes, in Product-NFL, are similar to Process-NFL ones (Section 3.2): \(\text{import}\) makes available the NF-Attribute definitions necessary in the NF-Attribute being defined; \(\text{components}\) defines the sub-attributes that compose the NF-Attribute; \(\text{contribution}\) defines the kind of contribution; and the \(\text{constraint}\) clause imposes constraints over the sub-attributes. Components in Product-NFL are expressed through typed value(s), as the attribute has to be precisely defined. The \(\text{predicate}\) defines constraints over the values assigned to components. Additionally, a common information associated to NF-Attributes, their units, is also defined in the attribute template. An example of the NF-Attribute \(\text{performance}\) in Product-NFL is shown on the right hand side.

The second template is defined for expressing the non-functional property itself. This template contains the \(\text{import}\) and \(\text{predicate}\) clauses. The first one has the same role as the \(\text{import}\) clause of the previous template. The second clause defines a predicate that imposes constraints over the values of components according to the requirement being defined.

### 4. CONCLUSION AND FUTURE WORK

This paper has presented a framework that defines how to treat with NFRs within the software development process. Parmenides defines two languages for describing NFRs, an integration strategy, a set of refinement rules and a mapping strategy. All these elements are introduced for dealing with NFRs from the early stages of development to the implementation. Product-NFL and Process-NFL are complementary languages\(^4\) that must be used in two distinct stages of the development. The integration strategy place together software architecture concepts and non-functional information, while the refinement rules define how the non-functional software architecture may be refined. In the last step of the development process, the mapping strategy guides how the abstract elements may be mapped into actual implementation elements.

The framework and its components have been tested to describe NFRs and to implement non-functional software architectures over EJB. In the former activity, Product-NFL has been tested for describing NFRs listed in \([2]\). The description of a great variety of NFRs was useful to indicate improvements in the language. Product-NFL was simplified and additional elements were included in the language in order to make it more expressive. For example, it was included the idea of “unit” (see Figure 4) as an important information to be present in the description of NFRs. Moreover, as a result of the use of Product-NFL, a catalogue has been created containing the Product-NFL description of a large number of NFRs.

Obviously, this proposal does not resolve all the problems related to the explicit treatment of NFRs. This is a very complex task that comprises essentially a stronger formalisation task, additional refinement rules, a better understanding of specific NFRs and further studies about the NFRs themselves. For example, it is essential to precisely define the meaning of compatibility among the NF-Realisations mentioned in Section 3.4. However, Parmenides is an effective step towards the explicit treatment of NFRs, as the framework proposes the integration of description, refinement and mapping tasks in a clear way. At the same time, the use of new concepts of software architecture and COTS components creates a realistic scenario for effectively dealing with NFRs. Additionally, the proposed languages are easier to be used if compared with logical notations and enable a more

\(^{3}\)This term is frequently used to refer to software packages that have been developed or are suitable for reuse.

\(^{4}\)We have already built syntax analysers for them using JavaCC.
concrete view of NFRs and their correlation.

As the treatment of NFRs in the literature is only on the birth, additional points have to be investigated from this framework. Firstly, the formalisation has to be extended, in the sense that it must cover all stages of the development. This task implies in the definition of the formal semantics of the Process-NFL and formalisation of the refinement rules. Secondly, the notations provided by the framework have to be used for describing further NFRs. It enables to create NFRs catalogues, specially using the Product-NFL. Finally, it has to be analysed the potentiality of the semi-automatic generation of Product-NFL from the Process-NFL description.

5. REFERENCES


