The Convenience for a Notation to Express Non-Functional Characteristics of Software Components

Xavier Franch

Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
c/Jordi Girona 1-3, 08034 Barcelona Catalonia (Spain)
Tel: (34) 3 4016965
Fax: (34) 3 4017014
Email: franch@lsi.upc.es

Abstract

Software systems are characterised both by their functionality (what the system does) and by their non-functionality (how does the system behave with respect to some observable attributes like performance, reusability, reliability, etc.). Both aspects are relevant to software development. However, non-functional issues have received little attention compared to functional ones. In this position paper we highlight the role of non-functionality, and we claim to have defined a notation to deal with them. We enumerate some design principles for such a notation, and then we make a proposal, which allows to define non-functional attributes of software, non-functional behaviour of components with respect to these attributes, and also non-functional requirements over implementations.

Keywords: Component, Non-Functional Attribute, Non-Functional Behaviour, Non-Functional Requirements.

Workshop Goals: learning; to highlight a particular problem not always perceived as such; to propose a concrete notation to deal with this problem; to receive feedback from the component programming community.

1 Background

Since my graduation in the late eighties, I have worked at the University teaching and doing research, and also being a partner in some granted projects. All of these activities have taken the concept of component as a prominent one, usually in the form of abstract data type.

About my research, I have been mainly involved in the study of formal specifications, remarkably in the algebraic framework, which was the former goal of my Ph.D. Later, I slightly switched the subject of my thesis to include non-functional aspects of software in the algebraic framework. The final result was a language (with completely defined formal semantics) to complement the usual algebraic specifications and imperative implementations of components with non-functional information appearing in both of them [Fra96].

On the other hand, my teaching activity has focused in data structures courses and also in programming-in-the-large with abstract data types. One of the main results of this activity is the publication of a book on data structures [Fra93]. As remarkable points of this book, I would like to mention that: data structures are defined just as means to implement abstract data types; and, I have included a whole chapter to study the design of new data structures to implement new abstract data types. Precisely, this last chapter emphasises the concept of reusable component as the main tool to build new software.

Last, I have participated in a pair of granted projects (the ICARUS Esprit project and a national one) both of them again focusing in formal specification of abstract data types. Recently, I have obtained a grant for a new project entitled "ComProLab: A Component Programming Laboratory" (outlined at [FBBR97]), leading a group of six researchers.

2 Position

2.1 Dealing with Non-Functional Aspects of Software

Software systems are characterised both by their *functionality* (what the system does) and by their *non-functionality* or quality[1] (how does the system behave with respect to some observable attributes like performance, reusability, reliability, etc.). Both aspects are relevant to software development. However, non-functional issues have received little attention compared to functional ones: there are a lot of formal specification languages and formal methods to deal with functionality of systems, but non-functionality is addressed by just a few approaches, often semi-formal or informal and limited in scope.

These approaches can be classified as *process-oriented* or *product-oriented*. Process-oriented approaches [MCN92, LS95] use non-functional information to guide the development of software systems. On the other hand, product-oriented approaches deal with non-functional issues by means of stating non-functional characteristics in the components themselves, being then possible to examine software products to check if they fall within the constraints of non-functionality. In this position

[1]We have rejected the word "quality" because there are some non-functional characteristics of software which are not related with the quality itself; for instance, the kind of user interface of a system, or the programming language used to write the code.

paper we are going to focus in the product-oriented side; however, it is important to remark that product-oriented and process-oriented techniques should be seen not as alternative but as complementary, both contributing to a comprehensive framework for dealing with non-functionality.

2.2 Why Non-Functionality is so Important?

The lack of non-functional issues in software components has some negative effects on many software development tasks.

- **Specification**. Non-functional characteristics of software remain hidden to the user and they only appear in (optional and informal) software documentation. Their absence leads to an unbalanced specification, where functional aspects are well covered with usual specification languages while non-functional ones do not exist.
- **Implementation**. The selection and/or development of the most appropriate (with respect to non-functional requirements) implementation for software modules cannot be assisted at all because of lack of information. As a result, the decisions to be taken during this process may be difficult and even incorrect.
- **Maintenance**. Changes in the system environment, modifications of existing software module implementations, and creation of new implementations require a new (by-hand) review of previously taken implementation decisions, without having available the non-functional information of the system, which is affected by these changes [FB97].
- **Reusability**. Software reuse cannot take non-functional issues into account. Thus, components selected by any functional-oriented reuse strategy may not fit into the non-functional requirements of the environment, hindering or even preventing their actual integration into the system.

2.3 Design Principles

Once we have advocated a notation for stating non-functional information in software components, we present here some design principles for such a notation.

- It should be complete. It should be possible to deal with non-functionality from different points of view; for instance, it is important to state non-functional properties of software components in specifications (e.g., "reliability of the component must be medium or high", "component must be portable from a PC platform to an UNIX system", etc.), but also to state non-functional characteristics of component implementations (e.g., "the insert and delete operations of the component take constant time").
- It should be as open as possible. Provided that non-functionality means different things for different people, it should be possible for each of them to choose the particular attributes of interest and to reject others; even more, there should exist means to group attributes and to form then non-functional libraries. Also, it could be useful to have the opportunity to bind some attributes just to some particular components, or even to just some parts (typically, operations) of components.

- It should be concise, clear and easy to use. Writing non-functional information should not be a difficult task, because this would conflict with the main idea behind them: their usefulness in software development. So, non-functional properties should be written as usual expressions (e.g., "reliability >= medium", "works-on(PC-platform) => works-on(UNIX)", etc.), and the same with implementations (e.g., "time(insert, delete) = O(1)").

- The notation must be a constituent part of software. This requirement highlights the role of non-functional issues in software development and it is achieved in many ways: non-functional information should be encapsulated in modules bound to software component definitions and implementations; the notation should be entirely formal, in the sense that its syntax and semantics are expected to be well-defined; and an algorithm should exist to verify that implementations fulfil non-functional specifications.

- It should fit to other software areas as much as possible. Although we are interested here in component programming, it would be nice to design a notation flexible enough to be used also in, say, information systems, knowledge-based systems, etc., with a few modifications. It should remain clear that a universal notation is impossible to get: for instance, in the component programming framework we can measure efficiency by the asymptotic behaviour of operations, while in information systems we are mainly interested in response time or throughput.

- It should be independent of the concrete specification and programming languages used to specify and code components[2]. The only restriction we put on those languages is modularity: software components must be really encapsulated in modules. Also, we require for every software component to have a single specification (at least, declaration of its public symbols -type or class name; procedures, attributes, methods of functions with their interface; etc.-) and possibly many implementations, each of them in a separate module.

- It should be complemented with tools. For instance, we have already mentioned the need for a tool able to verify if an implementation satisfies the properties concerning non-functionality that were stated. Also, it would be nice to have an algorithm able to select the better implementation of a component in every context where it is used. The existence of this algorithm would allow the whole software system to adapt to changes in the environment in an automatic manner, once the non-functional information were modified conveniently. Also, it would help component reusability and it would improve quality of design.

3 A Proposal: NoFun

In this section, we present the main issues of a notation called *NoFun* following the design principles enumerated in the last section. We classify non-functional information into three kinds:

- *Non-functional attribute* (short, *NF-attribute*): any characteristic of software which serves as a means to describe it and, possibly, to evaluate it. Among the most widely accepted [IEEE92, ISO91] we mention: time and space efficiency, reusability, maintainability, reliability and usability. In our approach, we let arbitrary attributes to appear; so, software components are

[2] Obviously, there can be a few changes to fit the notation into a particular pair of languages, most of them syntactic in nature [Fra97]

studied with respect to a particular set of NF-attributes; we say then that the component is *characterised* by this set.

- *Non-functional behaviour* of a component implementation (short, *NF-behaviour*): any assignment of values to the NF-attributes that characterise the implemented component.
- *Non-functional requirement* on a software component (short, *NF-requirement*): any constraint referred to a subset of the NF-attributes that characterise the implemented component.

The set of NF-attributes that characterise a component, together with their relationships (stated as NF-requirements), are declared in a *NF-specification module*, which is bound to the component specification. The NF-behaviour of an implementation is stated in a *NF-behaviour module*, bound to the implementation. Also, NF-behaviour modules will usually include NF-requirements for the software components imported by the implementation. Keeping non-functional information in separate modules gives full independence from the particular specification and programming languages used in the system.

3.1 Non-Functional Attributes

NF-attributes in NoFun are characterised by:

- Their **domain**. It fixes the set of valid values and operations. We have currently defined: boolean (e.g., error recovery), integer (e.g., degree of testing), real (e.g., response time), by enumeration (e.g., kind of user interface), string (e.g., programmer name) and asymptotic (time and space efficiency).

  NF-attributes must be declared in order to be known in components, except for asymptotic NF-attributes, which existence is inferred from the corresponding software component definition. More precisely, there are two asymptotic implicit NF-attributes, time(op) and space(op), for every public operation op, and another one space(t) for every public type t. Values of asymptotic NF-attributes are given in terms of some *measurement units*, which represent problem domain sizes and which must also appear in NF-specification modules.

- Their **kind**. NF-attributes can be *basic* or *derived*, depending on whether their value can be inferred from others or must be explicitly given in NF-behaviour modules. Derived NF-attributes are useful not only for avoiding redundant assignments, but also to provide a logical structure to the universe of NF-attributes.

  A derived NF-attribute P includes the following parts:

  - The list L of other NF-attributes that determine P's value.
  - A list of n guarded formulae of the form $C_i \Rightarrow P = E_i, 1 <= i <= n$, C_i being a boolean expression and E_i an expression yielding a value in P's domain; if $n = 1$, then C_i is optional. The meaning of a formula is: P equals E_i if the condition C_i holds. In order to have a well defined semantics, the union of the C_i must cover all possible cases and their pairwise conjunction must yield false.

- Their **scope**. We have mentioned that NF-attributes can be defined inside NF-specification modules; in this case, they are only know in the component associated with this specification.

Also, non-functional attributes may be defined in what we call *attribute modules*, which introduce closely-related and widely-applicable NF-attributes (appearing in many NF-specification modules, even in different software systems). In fact, attribute modules allow users to define their own libraries of NF-attributes which can be imported freely in software systems (of course, an attribute module may import other attribute modules).

Last, we have defined another kind of module, *system modules*, to introduce NF-attributes to be know in all the components of a whole software system. System modules may also import attribute ones, meaning that all the NF-attributes defined in these attribute modules are know in all the system. Attribute modules and system modules altogether make possible to define NF-attributes in a structured and easy manner.

We present next an example of definition of NF-attributes. An attribute module for the reliability attribute may include the definition of some auxiliary NF-attributes, some of them defined in the imported attribute modules: a boolean one *error_recovery*, and an integer one *test* with valid values from 0 to 5. NF-specifications may import this module in order to have this attribute as a measure of quality.

attribute module RELIABILITY

    imports ERROR_RECOVERY, TEST

    attributes

        boolean fully_portable

        enumerated ordered reliability (none, low, medium, high) **derived**

        depends on error_recovery, fully_portable, test

        defined as

            not error_recovery and not fully_portable => reliability = none

            not error_recovery and fully_portable => reliability = low

            error_recovery and not fully_portable => reliability = low

            error_recovery and fully_portable =>

                test in (0..1) => reliability = low

                test in (2..3) => reliability = medium

                test in (4..5) => reliability = high

end RELIABILITY

3.2 Non-Functional Behaviour

Once a component specification (both functional and non-functional parts) has been built, implementations for the component may be written. Each implementation for a given software component D should state its NF-behaviour with respect to the basic NF-attributes characterising D; values of derived NF-attributes are automatically computed. This assignment of values is encapsulated in a NF-behaviour module. In order for the implementation to be considered as correct, its NF-behaviour must fulfil all the requirements stated at the NF-specification (see next subsection).

In the general case, a component will be used in different software systems. In these systems, the NF-attributes characterising the component could be different. This situation requires multiple NF-behaviour modules to exist, each of them describing the NF-behaviour of the component in its corresponding context.

3.3 Non-Functional Requirements

NF-requirements are the means to state conditions on implementations of software components. Syntactically, they are usual boolean expressions enriched with some ad hoc constructs for non-functionality. Their purpose is to express relationships between NF-attributes and to represent the environment where implementations are to be inserted:

- Completing NF-specifications. NF-requirements are used in NF-specifications to state the conditions that every implementation of a software component must fulfil. In fact, they may appear at three different places: system modules, in which case the NF-requirement applies in all the components of the system; attribute modules, so that the NF-requirement is valid in every component importing this module; and NF-specification modules, to state a NF-requirement locally to a component.

- Relating measurement units. Efficiency is stated in components using different measurement units, but there has been no way to relate their value up to now. It seems natural to leave measurement units unrelated in the modules introducing them, because this yields components that can be reused in many contexts; however, it also seems convenient to relate the units once the components are considered as part of a particular software system. This kind of information is useful not only to complete non-functional specification of systems, but also to allow the evaluation of expressions involving different measurement units, which is essential in order to find out if an implementation satisfies this kind of NF-requirements.

- Fixing implementations of imported components. NF-requirements appearing in the NF-behaviour module bound to an implementation V in a system S state the conditions that the implementations of the software components imported by V must fulfil in S. In the general case, V will include a list of NF-requirements for every imported component; NF-requirements in the list are considered in order of appearance (which corresponds to the usual case of having requirements with different degrees of importance). As an alternative to the list, an implementation for a particular software component may be fixed directly by its name.

  Note that a single software component may be required in different ways at different places in the system due to the existence of different NF-requirements for it. Eventually, this will cause different implementations of the same component to coexist; this situation is supported by many programming languages (for instance, the O.-O. family using inheritance to represent the implementation relationship), although free interaction is usually restricted (see [Sit92, Fra94] for different proposals to avoid such restrictions).

4 Comparisons

As far as we know, there is no proposal for a language with the constructs proposed here, although many researchers have advocated for it [Jaz95, Sha84, Win90]. There are many non-formalised proposals [Mat84, LG86] the results of which are subsumed in our work. Also, [Win89] presents a case study to deal with boolean NF-attributes in an object-oriented framework; no other kind of properties are dealt with in her approach. An interesting proposal appears in [CZ90], which provides a framework to evaluate the design of software systems, the measurement criterion being the adequacy of implementations with respect to some non-functional requirements stated over a set of attributes. The requirements are stated as an array of weights over the properties and every attribute has a weight too; then, the evaluation of implementations results in a number and comparison is possible. However, the notation proposed in this work is not as general as that presented here; also, the proposal is not integrated into the software itself losing some of the advantages we have mentioned in the introduction.

On the other hand, [CGN94], [Sit94] and [SY94] provide a language to state program efficiency. [CGN94] aims to code generation from some high-level language constructs manipulating a relation data type; in the general case, there are many ways to generate this code and so information about efficiency is used to select the optimal translation. [SY94] focuses on program transformation: algorithms are refined using a library of components with pre-post functional specifications; when there are many components whose pre-post specification allows its inclusion in the algorithm being refined, efficiency is used to break the tie. Concerning [Sit94], it is the proposal closest to ours due to its definition in the component programming framework and also to the existence of special modules collecting some kind of non-functional information (constraints on efficiency in this case), although he focuses on software reusability and verification, which are two fields we have not yet addressed. Efficiency in [Sit94] is slightly more difficult to handle than in our work, because it is "tight" efficiency (an exact measurement of efficiency, more precise than the worst case we are considering here) and this often requires the definition of auxiliary models to express the time consumed by component operations. The proposal fits into a more exhaustive project, RESOLVE [Sit+94], which also includes a framework to allow switching of implementations of components [Sit92].

If we refer just to the notations used in the projects mentioned so far, none of them seem to be as powerful as NoFun, even considering just the subset of NoFun concerning efficiency. We would like to make sure of this fact in the workshop.

5 References

[CGN94] D. Cohen, N. Goldman, K. Narayanaswamy. "Adding Performance Information to ADT Interfaces". In *Proceedings of the Interface Definition Languages Workshop*, ACM SIGPLAN Notices 29(8), 1994.

[CZ90] S. Cárdenas, M.V. Zelkowitz. "Evaluation Criteria for Functional Specifications". In *Proceedings of 12th International Conference on Software Engineering* (ICSE), Nice (France), 1990.

[FB97] X. Franch, P. Botella. "Supporting Software Maintenance with Non-Functional Information". In *Proceedings 1st EUROMICRO Conference on Software Maintenance and Reengineering*, Berlin (Germany), 1997.

[FBBR97] X. Franch, P. Botella, X. Burgués, J.M. Ribó. "ComProLab: A Component Programming Laboratory". In *Proceedings of 9th Software Engineering and Knowledge Engineering Conference* (SEKE), Madrid (Spain), 1997.

[Fra93] X. Franch. *Data Structures: Specification, Design and Implementation.* Edicions UPC, col.lecció Politext, 30, 1993. Available in spanish.

[Fra94] X. Franch. "Combining Different Implementations of Types in a Program". In *Proceedings Joint of Modular Languages Conference* (JMLC), Ulm (Germany), 1994.

[Fra96] X. Franch. "Automatic Implementation Selection for Software Components using a Multiparadigm Language to state Non-Functional Issues". Ph.D. Thesis (advisor: Pere Botella), Universitat Politècnica de Catalunya, Barcelona (Catalunya, Spain), 1996. Available in català.

[Fra97] X. Franch. "Including Non-Functional Issues in Anna/Ada Programs for Automatic Implementation Selection". In *Proceedings of Ada-Europe'97*, London (U.K.), LNCS 1251, Springer-Verlag, 1997.

[IEEE92] IEEE Computer Society. *IEEE Standard for a Software Quality Metrics Methodology.* IEEE Std. 1061-1992, Institute of Electrical and Electronical Engineers, New York, 1992.

[ISO91] International Standards Organization. *Software Product Evaluation - Quality Characteristics and Guidelines for their Use.* ISO/IEC Standard ISO-9126, 1991.

[Jaz95] M. Jazayeri. "Component Programming - a Fresh Look at Software Components". In *Proceedings of 5th European Software Engineering Conference* (ESEC), Barcelona (Catalunya, Spain), LNCS 989, Springer-Verlag, 1995.

[LG86] B. Liskov, J. Guttag. *Abstraction and Specification in Program Development.* The MIT Press, 1986.

[LS95] D. Landes, R. Studer. "The Treatment of Non-Functional Requirements in MIKE". In *Proceedings of 5th European Software Engineering Conference* (ESEC), Barcelona (Catalunya, Spain), LNCS 989, Springer-Verlag, 1995.

[Mat84] Y. Matsumoto. "Some Experiences in Promoting Reusable Software". IEEE Transactions on Software Engineering, 10(5), 1984.

[MCN92] J. Mylopoulos, L. Chung, B.A. Nixon. "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach". IEEE Trans. on Software Engineering, 18(6), 1992.

[Sha84] M. Shaw. "Abstraction Techniques in Modern Programming Languages". IEEE Software, 1(10), 1984.

[Sit92] M. Sitaraman. "A class of programming language mechanisms to facilitate multiple implementations of the same specification". In *Proceedings 4th International Conference on Computer Languages*, IEEE Computer Society Press, 1992

[Sit94] M. Sitaraman. "On Tight Performance Specification of Object-Oriented Components". In *Proceedings 3rd International Conference on Software Reuse* (ICSR), IEEE Computer Society Press, 1994.

[Sit+94] M. Sitaraman (coordinator) et al. "Special Feature: Component-Based Software Using RESOLVE". ACM Software Engineering Notes, 19(4), Oct. 1994.

[SY94] P.C-Y. Sheu, S. Yoo. "A Knowledge-Based Program Transformation System". In *Proceedings 6th Conference on Advanced Information Systems Engineering* (CAiSE), Utrecht (Holland), LNCS 811, 1994.

[Win89] J.M. Wing. "Specifying Avalon Objects in Larch". In *Proceedings of Theory and Practice of Software Development*, Vol. 2, Barcelona (Catalunya, Spain), LNCS 352, 1989.

[Win90] J.M. Wing. "A Specifier's Introduction to Formal Methods". IEEE Computer 23(9), 1990.

6 Biography

Xavier Franch is associate professor at the Computer Science Department of the Universitat Politècnica de Catalunya (U.P.C.), Barcelona (Catalonia, Spain). He teaches courses of data structures, programming with abstract data types and introductory programming. He has written a book on data structures in 1993, which is a basic reference at many spanish universities. He currently leads a group of six people involved in many aspects of component programming: definition of a notation to state non-functional issues of components, automatic synthesis of non-functional attributes, integration of functional specifications and imperative code and definition of software processes. He received a Ph.D. in Computer Science from the U.P.C. in 1996.