Systematic Formulation of Non-Functional Characteristics of Software[1]

Xavier Franch
franch@lsi.upc.es
Dept. Llenguatges i Sistemes Informàtics (LSI)
Universitat Politècnica de Catalunya (UPC)
c/Jordi Girona 1-3 (Campus Nord, C6), 08034 Barcelona, Catalonia (Spain)
FAX: 34-3-4017014. Phone: 34-3-4016965

Abstract

This paper presents NoFun, a notation aimed at dealing with non-functional aspects of software systems at the product level in the component programming framework. NoFun can be used to define hierarchies of non-functional attributes, which can be bound to individual software components, libraries of components or (sets of) software systems. Non-functional attributes can be defined in several ways, being possible to choose a particular definition in a concrete context. Also, NoFun allows to state the values of the attributes in component implementations, and to formulate non-functional requirements over component implementations. The notation is complemented with an algorithm able to select the best implementation of components (with respect to their non-functional characteristics) in their context of use.

Key words: component programming, non-functional requirements.

1. Introduction

1.1. Motivation

Software systems can be characterised both by their *functionality* (what the system does) and by their *non-functionality* or quality[1] (how the system behaves with respect to some observable attributes like performance, reusability, reliability, etc.). Both aspects are relevant to software development; in this paper, we are going to focus on the study of non-functionality.

Approaches to non-functionality can be classified as *process-oriented* or *product-oriented*. Process-oriented ones use non-functional information to guide the development of software systems. There are some widespread approaches in the information systems area [3, 14] as well as in knowledge-based systems [13]. On the other hand, product-oriented approaches deal with non-functional issues from the evaluation point of view: software products may be examined to check if they fall within their constraints of non-functionality. As stated in [14], 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.

A natural way to facilitate the product-oriented approach is to define a notation aimed at stating non-functional requirements of software in the software itself. Although many researchers have pointed out the convenience of this notation [2, 11, 16, 18, 21], there seem only to be semi formal (even informal) or limited (with respect to the kind of non-functional information managed) proposals in the software community. The lack of such a comprehensive and formally defined language 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 software documentation. Their absence leads to unbalanced specifications, 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 automated at all because of lack of precise information. As a result, the decisions to be taken during this process may be difficult and even incorrect.

[1] This work is partially supported by the spanish project TIC97-1158 (from the CICYT program).

[2] 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.

- 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.
- Reusability. Software reuse cannot take non-functional issues into account. Thus, modules 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.

1.2. The framework

In this paper, we focus on the *component programming* field as defined in [11, 19], which is characterised by the existence of software components with: 1) a *specification* introducing the public symbols of the component together with the statement of their behaviour; and 2) many *implementations*, each of them designed to fit a particular context of use, depending on its non-functional characteristics (efficiency of operations, reliability, etc.). We consider that every component implementation is kept in a separated module, as well as its specification.

Beside software components, we consider also other kinds of units that will be of interest for stating non-functionality:

- Libraries of reusable software components. Grouping some subject-related components.
- Software systems. Combination of software components, using also some libraries.
- Clusters. Sets of software systems which are related by some criteria (subject, software team, etc.).

The physical implementation of these units (i.e., their mapping to concepts like files, directories, and user working space) is not of interest for our work, although it should be considered when adopting our proposal to a particular environment.

1.3. The proposal

We present a notation called *NoFun* aimed at binding non-functional information to software modules in the component programming field. This information is classified into three kinds:

- *Non-functional attribute* (short, *NF-attribute*): any attribute of software which serves as a way to describe it and possibly to evaluate it. Among the most widely accepted [9, 10] we can mention: time and space efficiency, reusability, maintainability, reliability and usability. In our approach, we allow arbitrary identification and definition of NF-attributes.
- *Non-functional behaviour* of a component implementation (short, *NF-behaviour*): any assignment of values to the NF-attributes that are in use in the implemented component.
- *Non-functional requirement* on a software component (short, *NF-requirement*): any constraint referred to a subset of the NF-attributes that are in use in the component.

In the rest of the paper, we are going to present the constructs of NoFun for stating these three kinds of information, and their organisation in modules.

2. Non-Functional Attributes

In addition to their name, NF-attributes have the following characteristics in NoFun:

- They belong to a *domain*, which fixes the set of valid values and operations.
- Their are classified of one of two *kinds*, basic or derived.
- They have a *scope*, which determines the components in which they are in use.
- They can be *bound* either to whole components or to individual operations.
- They can have *multiple definitions*.

Except for the scope, these characteristics appear when the NF-attribute is defined inside a *NF-attribute module*; a single module may define more than one NF-attribute. In case of multiple definitions, each of them will appear at different modules, see 2.5.

NF-attribute modules may import others; as a particular case, libraries of NF-attributes may be simulated by a NF-attribute module importing those ones of interest.

2.1. Domains

We have identified the following standard domains:

- Boolean. To represent software attributes which just hold or fail, such as error recovery. Usual boolean operations may be used.
- Integer, real. To introduce software attributes which can be measured, such as the degree of usability of a component (with an integer number), or the maximum response time of an operation (with a real number). Lower and upper limits of these attributes may be declared. Usual arithmetic operations may be used.
- Enumeration. To deal with software attributes which can be classified into various categories, such as kind

of user interface (icons, command language, etc.). The set of valid values should be declared. The values may be declared as ordered (from left to right), and so some operators (<, >, <=, >=, *max* and *min*) become available. In any case, comparison of values is possible.

- String. To declare software attributes which can be labelled, such as the name of the programming language used to implement the component. Strings can be compared.
- Mapping. To define software attributes which value depend on others. For instance, we can declare a attribute for full portability of implementations as a mapping from an enumeration attribute (the platform: UNIX, Windows-95, etc.) to booleans. The basic operation on mappings is application to some value.

Also, we have added a specific domain, the domain of efficiency, to measure the cost of individual operations and type representations. The NF-attributes concerning efficiency need not be explicitly declared; their existence is inferred from the corresponding software component definition. More precisely, there are two implicit NF-attributes, time(*op*) and space(*op*), for every public operation *op*, and an implicit NF-attribute space(*t*) for every public type *t*. Values of this kind of NF-attributes are given in terms of some *measurement units*, which represent problem domain [illegible] reason for keeping apart this domain from [illegible] is that the behaviour of operations is [illegible] the efficiency expression power(*n*, 2) + 5 [illegible] equal to power(*n*, 2), and also the equality 5 *n* = 12 *n* + log(*n*) holds; in both cases, *n* is a measurement unit.

2.2. Kinds

NF-attributes may be classified as *basic* or *derived*, depending on whether their value can be computed from others or not. For instance, in 3.1 we will define the reliability of a component as a derived NF-attribute, its value depending (among others) on a basic NF-attribute stating if the component has error recovery or not. In the case of basic NF-attributes, implementation of components will assign values to them; in the case of derived ones, values will be computed automatically.

A derived NF-attribute *P* includes the following parts:

- The list *L* of other NF-attributes (which may also be derived) that determine *P*'s value.
- A list of guarded formulae of the form C_i => $P = E_i$, $1 \le i \le n$, 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. As a correctness condition, the union of the C_i must cover all possible cases and their pairwise conjunction must yield false.

2.3. Scope

Concerning its scope, NF-attributes may be in use in those kind of units identified in 1.2:

- Individual components. For instance, the NF-attribute "kind of user interface" should be defined just for those components interacting with the environment.
- Libraries of reusable components. For instance, (floating point) accuracy is a NF-attribute of interest in mathematical libraries.
- Software systems. For instance, a project involving heavy network communication or access to remote, large databases may define a NF-attribute for reliability of physical media.
- Clusters. This is the way that can be used by a company or by a software team to define the set of NF-attributes that they consider relevant in all their projects.

The scope is fixed by writing the name of the NF-attribute module in the corresponding software unit (component, library, system or cluster); in other words, we need to annotate these units. Note that the scope of all the NF-attributes introduced in this module is the same. All the NF-attributes imported in a NF-attribute module *M* must be known in the scopes which *M* is bound to; if not, they are implicitly added to the scopes that miss them.

2.4. Bindings

Although we usually think of NF-attributes as bound to whole components, it may be the case of having others referring to individual operations; this is the case of the efficiency NF-attributes as defined in 2.1. This is why NF-attributes should be bound to components or (subsets of) operations, being the first case the default (except for the efficiency case).

As a special case, a NF-attribute could be bound both to a component and to some operations; then, the component-bound NF-attribute should be defined as derived, in terms of the operation-bound ones. For instance, reliability of a component could be defined in terms of the reliability of its operations.

Some remarks must be made concerning bindings and derived NF-attributes. Let *P* be a derived NF-attribute and let Q_1, ..., Q_n be the NF-attributes upon which *P* depends:

- If *P* is bound to a component *C*, then all of Q_1, ..., Q_n must also be bound to *C*.

- If P is bound to an operation *op*, then all of Q_1, ..., Q_n must be bound to either *op* or the component defining *op*. In other words, a NF-attribute bound to a component may be used as bound to an operation if the context requires it.

2.5. Multiple definitions

It is a fact that there does not currently exist a universal repository of NF-attributes recognised as such in the requirements engineering community. Furthermore, for those ones that could be admitted as such, we can find different definitions in different papers, standards or projects. This is why we have decided to allow multiple definitions of NF-attributes. When using a multiple-defined NF-attribute, a particular definition should be chosen in every scope where the NF-attribute is in use.

Multiple definitions yield the following modular structure:

- There must be a common NF-attribute module containing the name, domain and binding of the multiple defined NF-attribute(s). Also, if there are more NF-attributes to be put in the module with a single definition, they can be included in this module.
- There must be a different NF-attribute module for every different combination of definitions of the multiple-defined NF-attributes[3].

It follows from this description that, regardless of the particular definition, the domain and binding of multiple-defined NF-attributes must be the same. This seems natural to assume because we think that users of a NF-attribute should be able to reason about it independently of the chosen definition; this independence would not be possible if, say, reliability were defined with different domains at different modules (e.g., as an integer and by enumeration with values {*high*, *medium*, *low*}).

3. Examples

We give here an example of definition of two particular NF-attributes: reliability and reusability. We are going to develop in detail the first case, while the second one will be just outlined.

3.1. Reliability

We present here a simple and naïve (for the sake of brevity) definition of reliability of implementations. In despite of this simplicity, it should remain clear that NoFun is able to handle more complicated and precise definitions with its constructs, close to the ones presented in the example.

We structure the definition in various modules. First of all, we introduce a NF-attribute to state if an implementation presents some kind of error recovery or not. We define this component-bound attribute in terms of another NF-attribute with the same name, bound to all the operations of the component: we state that a component implementation has an error recovery mechanism if and only if all its operations have error recovery. Note the use of some built-in symbols (*all_ops*) and predicates (*for all*), with an obvious meaning.

```
attribute module ERROR_RECOVERY
  attributes
    boolean error_recovery bound to all_ops
    boolean error_recovery
        bound to components derived
        depends on error_recovery(all_ops)
        defined as
            error_recovery =
                for all op in all_ops it holds
                    error_recovery(op)
end ERROR_RECOVERY
```

Fig. 1: A definition of a NF-attribute for error recovery.

Next, we introduce another NF-attribute for test. We decide to measure testing of individual operations with an integer from zero to five. Then, we introduce a derived, component-bound NF-attribute for testing of implementations. We provide two different definitions of this attribute, each one defined in a different module; both modules are linked to the one introducing the NF-attribute with a "refines" construct. The definitions use some built-in functions, which have been included in NoFun due to their usefulness in defining various NF-attributes. The first module, the pessimistic one, defines the testing value of the implementations as the minimum of the testing values of its operations; the second definition computes as result the arithmetic mean of the testing values of the operations.

Note that, in any case, it is not obvious how do we get the value for the testing NF-attributes bound to operations. In spite of its importance, this is not a subject covered by our work; our goal is providing a mean to represent these values whatever the way of getting them is.

Finally, we introduce the NF-attribute of interest, *reliability*, defined in terms of error recovery, test and a new NF-attribute, *fully_portable*, which will be true when an implementation uses just standard constructions of the corresponding programming language. The last four lines are an abbreviation: the condition in line (*) must be and'ed with the conditions in the last three lines.

[3] This is why it seems natural to include just one multiple-defined NF-attribute in a module.

```
attribute module TEST
  attributes
    integer test [0..5] bound to all_ops
                (* 0 to 5: increasing degree of testing *)
    integer test [0..5] bound to components
        derived
end TEST

attribute module TEST_BY_MIN
  refines TEST
  attributes
    integer test [0..5]
      depends on test(all_ops)
      defined as test = min(test, all_ops)
end TEST_BY_MIN

attribute module TEST_BY_MEAN
  refines TEST
  attributes
    integer test [0..5]
      depends on test(all_ops)
      defined as
              test = sum(test, all_ops) div #all_ops
end TEST_BY_MEAN
```

Fig. 2: Multiple definitions for testing.

```
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
        error_recovery and not fully_portable =>
                    reliability = low
        not error_recovery and 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
```

Fig. 3: A definition for reliability.

3.2. Reusability

The purpose of this example is to study the suitability of our approach for a more realistic and detailed proposal of NF-attribute, reusability, as done by Caldiera and Basili in [1].

Caldiera and Basili identify four factors that have influence on reusability: volume, cyclomatic complexity, regularity and reuse frequency, and then they provide a formula for each of them. We could encapsulate each of the four factors, together with the atoms that appear in its formula, in an individual NF-attribute module (each atom yielding a basic NF-attribute); as far as volume and regularity share some common atoms (values coming from the Halstead Software Science Indicators), we can encapsulate them in another module. Finally, we need a sixth module for the NF-attribute of interest, reusability. In fact, we could decide to give multiple definitions for reusability combining the four factors in different ways, depending in the context; the number of NF-attribute modules will then increase accordingly, as it happened with the *test* attribute in 3.1.

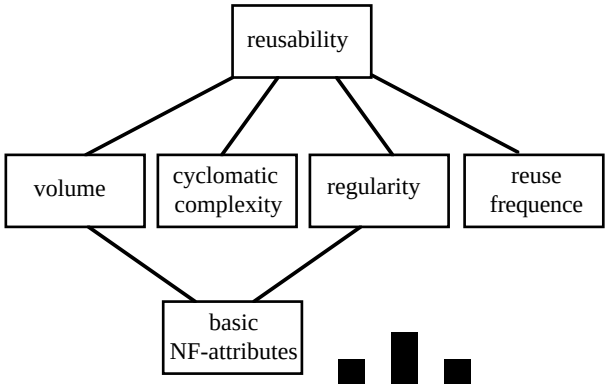

Fig. 4: Hierarchy of NF-attribute modules for reusability.

We show here the definition of the first two factors:

- Volume = $(N_1 + N_2)$ $\log(\eta_1 + \eta_2)$, being N_1 and N_2 the total count of all usage of operators and operands in the implementation, and η_1 and η_2 the total number of different operators and operands used in the implementation.
- Cyclomatic complexity = $e - n + 2$, being e and n the number of edges and nodes of the control-flow graph of an operation. In fact, this formula refer to the cyclomatic complexity of just one operation, and then we should compute the value for the component with the expression *sum(cyclomatic_complexity, all_ops)*, to be assigned to a component-bound NF-attribute (the one of interest).

We remark that, except for the use of Greek letters and subindexes, the formulae are valid expressions in NoFun. Also, it is important to note that the basic NF-attributes that appear in the formula can be computed in an automatic manner from code.

4. Non-Functional Behaviour

Once a component specification has been built, implementations for it may be written. Each implementation V for a given software component D should state its NF-behaviour with respect to the basic NF-attributes that are in use in D; values of derived NF-attributes are automatically computed. To keep non-functional informa-

tion apart from code, this assignment of values is encapsulated in what we call a *NF-behaviour module.*

In the general case, a component will be used in different software systems. In these systems, the attributes that are in use in the component could be different, and all of them should appear in the NF-behaviour module.

For instance, the behaviour for an implementation *IMP_LIBRARY_1* for a *LIBRARY* component in a *LIBRARY_MANAGEMENT* software system may look like the module in fig. 5. We are assuming that, in this system, *LIBRARY* is in the scope of the *RELIABILITY* NF-attribute module; so, it is necessary to give values to the basic NF-attributes introduced in this module, as well as to the implicit efficiency NF-attributes (as explained in 2.1). Concerning the first ones, we state that: all the operations of the component have error recovery; the implementation does not use non-standard language features; and its operations have a testing value of 4 except for the *check_out* operation. Concerning efficiency, we remark the use of arithmetic-like operators and measurement units (for instance, *n_members*, to represent the number of members of the library). With this assignment, and assuming that we have chosen the *TEST_BY_MIN* definition for *TEST*, the value of the (component-bound) derived properties are: *error_recovery* = true, *test* = 2 and *reliability* = *medium*.

```
behaviour module for IMP_LIBRARY_1
  behaviour
    error_recovery(ops(LIBRARY)); fully_portable
    test(ops(LIBRARY)) = 4
                    except for test(check_out) = 2
    time(list_all_members) = n_members
    time(check_out) = log(n_books)
  …
end IMP_LIBRARY_1
```

Fig. 5: Non-functional behaviour of an implementation for a LIBRARY component.

5. Non-Functional Requirements

Implementations of software components will usually import other components (to represent some types and/or to code the operations). To consider an implementation *M* complete, it is necessary to choose particular implementations for these imported components. We advocate here that the selection of the implementation for a component *C* imported in *M* should be done by comparing the NF-behaviour of *C* implementations with the NF-requirements stated over *C*; these NF-requirements modelise the context of use of *C* and will be expressed using NoFun too.

NF-requirements will be in fact organised as a list such that they 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.

For instance, let us suppose that *LIBRARY* uses two components *LIST* and *SET* to compose lists and sets of books, members, etc. Then, the implementation *IMP_LIBRARY_1* could state as NF-requirement over *SET* the following one: first, implementation must be as reliable as possible; next, the cost of insertions and removals must be constant; last, set intersection should be as fast as possible. Concerning *LIST*, the particular implementation *ORDERED_LIST* is directly selected.

```
behaviour module for IMP_LIBRARY_1
  behaviour
    ... as before
  requirements on SET: max(reliability)
                       time(put, remove) = 1
                       min(time(intersect))
                  on LIST: implemented with
                                ORDERED_LIST
end IMP_LIBRARY_1
```

Fig. 6: Non-functional requirements over imported components appearing in a LIBRARY implementation.

In this example, the NF-requirements have been stated locally in a component implementation. Also, it is possible to state NF-requirements bound to libraries, software systems or clusters. So, a company may represent its preferences in cluster-bound NF-requirements (for instance, requiring maximum reliability and full portability to UNIX platforms), which can be further constrained in systems and libraries, and being NF-behaviour modules the place to state local constraints, as in the example. We consider that NF-requirements in clusters have precedence over the ones in individual software systems, and these ones are also more prioritary than the other two.

As an alternative to the statement of NF-requirements for particular components, global NF-requirements can be formulated, affecting all the components in a cluster, library or system, or all the imported components in a component implementation. An example could be requiring a certain degree of reliability to all the components in a software system. Global NF-requirements take precedence over particular ones.

Note that using the full capabilities of NoFun, 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 [7, 17] for different proposals to avoid such restrictions).

6. Conclusions

We have presented NoFun, a language to state non-functional issues of software systems at the product level in the component programming framework. The language allows to declare non-functional attributes of software, to give values to these attributes in component implementations, and to formulate non-functional requirements in terms of these attributes. Non-functional information may be bound to various kinds of software units (components, libraries, systems and clusters) by means of annotations and special modules. In this paper, our goal has been to give an exhaustive presentation of the language capabilities, relegating the formal aspects, in order to convince the reader of the usefulness of the proposal.

We consider that the salient features of our approach are:

- The language provides a mean to formulate non-functionality in a precise way, different from the usual case (natural language). There is a lot of work done in studying non-functional attributes, defining metrics, and so on, but we think there is a lack of notations to express the concepts arising in the field. A notation such as NoFun provides then a common framework in which people can formulate, analyse and compare their proposals about non-functionality. We have represented in this paper a measure for reusability as formulated in [1], and we have developed also other proposals [5, 12].
- As far as NoFun has a well-defined syntax and semantics (not detailed here), we have been able to use it as a basis for building an algorithm to select component implementations in an automatic way, by evaluating them with respect to some non-functional requirements that modelise their context of use. We think that this particular point distinguishes our approach from others.
- The combination of both NoFun and the implementation selection algorithm can be an aid to software specification, design, reusability and maintenance. Concerning specification, we can complement usual functional specifications with non-functional aspects. Design is enhanced by having more detailed information available, and by using the algorithm to choose implementations. Reusability methods can be refined using non-functional characteristics to choose between functional-equivalent components obtained by retrieval in libraries of reusable components. Last, maintenance due to changes on non-functional aspects of systems can also benefit by automating the change of implementations as others become more appropriate [6].
- Concerning the power of the language, we would like to remark that it presents many features which are necessary to modelise non-functionality in a proper way: 1) non-functional attributes may be defined in more than one way; 2) they can be bound either to components or to operations[4] (or both); 3) they can have different scopes; 4) non-functional requirements may be ordered with respect to their relative importance.
- Our proposal can be adapted to classical modular programming languages [8]. We just require them to encapsulate components in modules. Also, we require 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 in a separate module. These requirements are satisfied by a huge class of languages.

As future work, we are currently addressing to automatic synthesis of values of NF-attributes in implementations. This is to say, we have provided no means in our proposal to compute the value of a specific basic NF-attribute from the code of the implementation; we are only able to calculate the value of derived NF-attributes from the corresponding formula. Note that if full automatic synthesis were carried out, NF-behaviour modules could disappear. However, it must remain clear that there are many NF-attributes whose values do not seem to be easily computable from code; an example is the *test* property used in this paper.

There are many approaches for defining a language to state non-functionality, but as far as we know they are limited in scope. They mainly address to many facets of efficiency: asymptotic efficiency [20], efficiency of queries in relational structures [2], tight efficiency [18] and real-time efficiency [15]. The last two approaches resemble ours in the sense that they define a grammar to formulate efficiency. Also, [18] introduces modules to encapsulate some kind of non-functional information.

Concerning automatic selection of implementations, we mention [4] as an approach close to ours, providing a framework to evaluate the design of software systems, the measurement criterion being the adequacy of

[4] We are currently considering the possibility to allow bindings to libraries, systems and clusters.

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. Again, the notation proposed in this work is very restricted compared to ours; also, the proposal is not integrated into the software itself losing some of the advantages we have mentioned.

References

[1] G. Caldiera, V.R. Basili. "Identifying and Qualifying Reusable Software Components". IEEE Computer, 24(2), 1991.

[2] 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.

[3] L. Chung, B.A. Nixon, E. Yu. "Using Non-Functional Requirements to Systematically Support Change". In *Proceedings of Second International Symposium on Requirements Engineering (ISRE)*, York (England), 1995.

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

[5] R.G. Dromey. "A Model for Software Product Quality. IEEE Transactions on Software Engineering, 21(2), 1995.

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

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

[8] X. Franch. "Including Non-Functional Issues in Anna/Ada Programs for Automatic Implementation Selection". *Proceedings of Ada-Europe'97. International Conference on Reliable Software Technologies*, London (U.K.), LNCS 1251, 1997.

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

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

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

[12] S.E. Keller, L.G. Kahn, R.B. Panara. "Specifying Software Quality Requirements with Metrics". IEEE Computer, 1990.

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

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

[15] R.H. Pierce *et al.* "Capturing and verifying performance requirements for hard real-time systems". In *Proceedings International Conference on Software Reliable Technologies*, London (England), LNCS 1251, Springer-Verlag, 1997.

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

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

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

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

[20] P.C-Y. Sheu, S. Yoo. "A Knowledge-Based Program Transformation System". In *Proceedings Sixth CAiSE*, Utrecht (Holland), LNCS 811, 1994.

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