RR/LSI-97-14-R

NoFun: A Notation to State Non-Functional Specifications at the Product Level

Xavier Franch

Report LSI-97-14-R

NoFun: A Notation to state Non-Functional Specifications at the Product Level

Xavier Franch
franch@lsi.upc.es
Dept. Llenguatges i Sistemes Informàtics (LSI)
Universitat Politècnica de Catalunya (UPC)
Pau Gargallo 5, 08028 Barcelona, Catalunya (Spain)
FAX: 34-3-4017014. Phone: 34-3-4016965

Abstract

This paper presents *NoFun*, a comprehensive and formally-defined notation to deal with non-functional specifications in software systems at the product level in the component programming framework. This notation is intended to complement the existing specification languages which focus on the functional behaviour of programs, and it is independent of the specification and programming languages used to develop the system. More precisely, NoFun is used to define both the set of non-functional properties which characterise a software component in the system and, for every implementation of this component, its non-functional behaviour as well as its non-functional requirements on other imported components. Although the proposal is developed in the component programming framework, we think that the results may be generalised to any other field that must take non-functional issues into account.

Key words: component programming, non-functional specifications, non-functional requirements.

1 Introduction

Software systems are characterised both by their functionality (what the system does) and by their non-functionality (how the system behaves 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; they are addressed by just a few approaches, which can be classified as *process-oriented* or *product-oriented.*

Process-oriented approaches use non-functional information to guide the development of software systems. The most widespread one was proposed by Mylopoulos *et al.* [MCN92, CNY95, and others] and it was developed in the information systems area. Also, some work has been done on knowledge-based systems, notably in the MIKE system [LS95], whose main ideas are close to those of Mylopoulos *et al.* On the other hand, product-oriented approaches deal with non-functional issues from the evaluation point of view: software products are examined to check if they fall within the constraints of non-functionality. As stated in [MCN92], 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-functional specifications and non-functional requirements.

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 [Sha84, Win90, MCN92, CGN94, Sit94, Jaz95],

there seem only to be 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 (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 automated 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.
- 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.

To facilitate these tasks, we present a notation called *NoFun* to bind non-functional information to software modules in the component programming field. This information is classified into three kinds: declaration of non-functional properties, statement of non-functional behaviour, and statement of non-functional requirements. The notation does not depend on the languages used to specify and implement the components, because non-functional information is actually encapsulated in separated modules.

The rest of the paper is structured as follows: in section 2 we present the framework of our proposal; section 3 shows how non-functional properties can be introduced using NoFun; sections 4 and 5 are devoted to the statement of non-functional behaviour and non-functional requirements; section 6 outlines the key points with respect to genericity and inheritance in software components; finally, section 7 gives the conclusions.

2 The Framework

In this paper, we focus on the *component programming* field as defined in [Sit+94, Jaz95], 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*, most of them using classical data structures like graphs, lists, hash tables and trees, whose results concerning non-functionality are well known. Every implementation is designed to fit a particular context of use.

We classify non-functional information into three kinds:

- *Non-functional property* (short, *NF-property*): any attribute 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 also let more specific attributes appear.

We have not defined a fixed catalogue of NF-properties. Instead, software components are studied with respect to a particular set of NF-properties; 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-properties that characterise the implemented component.
- *Non-functional requirement* on a software component (short, *NF-requirement*): any constraint referred to a subset of the NF-properties that characterise the component.

The set of NF-properties 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. This specification may import one or more *property modules*, which are used to introduce closely-related and widely-applicable NF-properties (appearing in many NF-specification modules, even in different software systems). In fact, property modules allow users to define their own libraries of NF-properties which can be imported freely in software systems.

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.

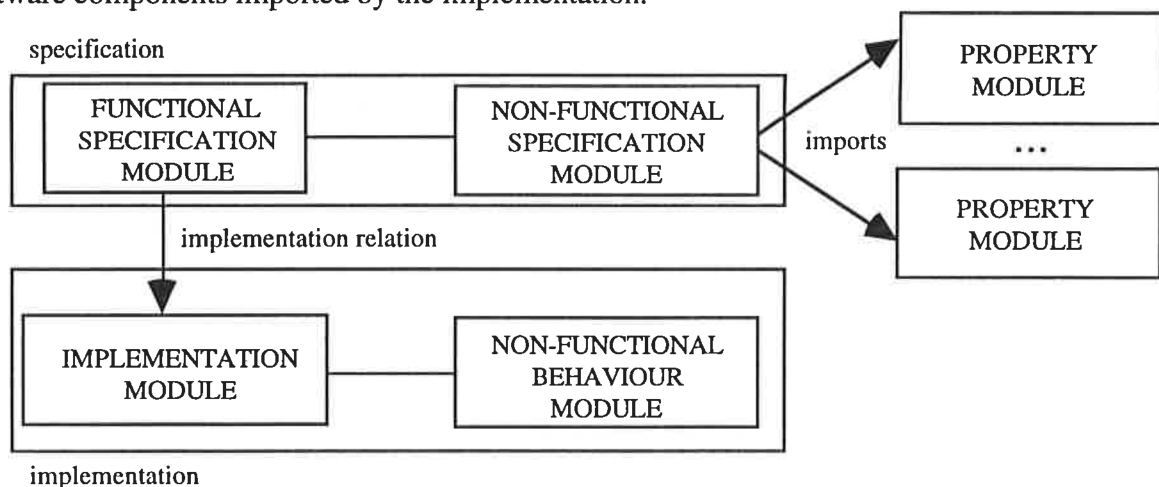

Fig. 1: Organisation of non-functional information into modules.

3 Declaration of Non-Functional Properties

3.1 Domains of NF-properties

NF-properties may be of many different kinds, depending on the domain of their values.

- Boolean. To represent software attributes which just hold or fail, such as full portability.
- 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 response time of an operation (with a real number). Lower and upper limits of these NF-properties may be declared.
- Enumeration. To deal with software attributes which can be classified into various categories, such as the kind of user interface (icons, command language, etc.). The set of valid values is expected to be declared. The values may be declared as ordered (from left to right), and so some operators (<, >, <=, >=, *max* and *min*) will be available.

- String. To declare software attributes which can be labelled, such as the name of the programmer of a component.
- Asymptotic. For measuring the worst-case asymptotic cost of individual operations or type representations, using the big-Oh asymptotic notation as defined in [Bra85] (set of functions). These asymptotic NF-properties need not be explicitly declared; their existence is inferred from the corresponding software component definition. More precisely, there are two asymptotic implicit NF-properties, time(*op*) and space(*op*), for every public operation *op*, and an asymptotic implicit NF-property space(*t*) for every public type *t*.

Values of asymptotic NF-properties are given in terms of some *measurement units*, which represent problem domain sizes and which must also appear in NF-specification modules. These values will involve a few big-Oh specific operators, such as power, logarithm and the like.

3.2 Derived NF-properties

NF-properties may be classified as *basic* or *derived*, depending on whether their value can be computed from others or not. In the case of basic NF-properties, implementation of components will assign values to them; in the case of derived ones, values will be computed automatically.

Derived NF-properties P include the following parts:

- The list L of other NF-properties that determine P's value.
- A list of n guarded formulae of the form $C_i \Rightarrow P = E_i$, $1 \leq i \leq n$, 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.

Derived NF-properties may eventually depend upon other derived ones, yielding a directed graph (which must be acyclic) representing relationships between those NF-properties. They are useful not only for avoiding redundant assignments in implementations, but also to provide a logical structure to the set of NF-properties relevant to software systems.

3.3 An example

We give here an example of definition of many NF-properties characterising a software component. First of all, we define property modules for two sets of related NF-properties (see fig. 2). The first, *CREATION_ISSUES*, groups information about the date of creation and the person responsible for an implementation (remember that non-functional information always refers to implementations of components); there is a derived property to state if this person does not belong to the company staff. The second one, *RELIABILITY*, is mainly devoted to introducing a derived property, *reliability*, defined in terms of three more basic ones, one of them (*external_responsible*) being imported from *CREATION_ISSUES*. The last four lines are an abbreviation: the condition in line (*) must be and'ed with the conditions in the last three lines.

We remark that the definition given for *reliability* is an arbitrary one: other specifiers may provide different metrics (possibly involving other NF-properties); moreover, different property modules could coexist defining *reliability* in different ways, and then their use in different contexts would be up to the specifier.

```
property module CREATION_ISSUES
   properties
      string responsible_name
      numerical month [1..12], year      (* date of delivery *)
      boolean external_responsible derived
         depends on responsible_name
         defined as external_responsible = not responsible_name in ["franch", "jones", "smith"]
end module

property module RELIABILITY
   imports CREATION_ISSUES
   properties
      boolean fully_portable
      integer test [0..5]    (* 0: not tested; 1 to 5: increasing degree of testing *)
      enumerated ordered reliability [none, low, medium, high] derived
         depends on external_responsible, fully_portable, test
         defined as
            external_responsible and not fully_portable => reliability = none
            external_responsible and fully_portable => reliability = low
            not external_responsible and not fully_portable => reliability = low
   (*)      not external_responsible and fully portable =>
                    test in [0..1] => reliability = low
                    test in [2..3] => reliability = medium
                    test in [4..5] => reliability = high
end module
```

Fig. 2: Two general-purpose property modules.

Next, we define a *LIBRARY* software component, to keep track of the set of books and members of a library, as well as the check outs of books by members. As stated above, there are some implicit asymptotic NF-properties coming from the public operations and type representations. In addition to them, we import the NF-properties declared in the property modules appearing in fig. 2 and we introduce a new boolean NF-property, private to this component, to state if implementations provide error recovery or not. Finally, we introduce four measurement units for stating efficiency.

```
non-functional specification module for LIBRARY
   imports RELIABILITY, CREATION_ISSUES
   properties boolean error_recovery
   measurement units n_books, n_members, n_book_copies, n_book_copies_borrowed
end module
```

Fig. 3: Declaration of non-functional properties for a LIBRARY component.

3.4 Bindings

Note that there is an important difference between NF-properties concerning efficiency and the others: we talk about time and space efficiency of individual operations while the other

NF-properties refer to whole components. Sometimes, however, it will be useful also to define NF-properties different from efficiency at the operation level and this is the reason for introducing the concept of binding: every NF-property declared in a NF-specification must be bound to the whole component or to a subset of its operations. In the last case, different values may be assigned to the NF-property in the chosen operations.

We remark that binding a NF-property to a component or an operation is again up to specifiers; for instance, we can talk about reliability of whole components or reliability of concrete operations. As a particular situation, a single NF-property may be bound both to components or operations in different places of the system, or even to an operation and the component defining the operation, in which case the component-bound NF-property will usually derive from the operation-bound one.

Given a NF-property P, its kind of binding may be established in three different places:

- System modules (see below).
- The property module introducing P, if this is the case. Then, the binding is valid in all the NF-specifications importing P. If all the NF-properties defined in a property module are bound to the same unit (component or operation), a single key word in the header is enough.
- The NF-specification of a component M. In this case, the binding is valid just in the component or the operation, depending on the exact kind. In the case of binding to operations, a set of them should be listed. Also, when P has been bound to operations in a property or system module S, the NF-specification should state the particular operations which P is bound to, except for the case of P having been bound to all operations in S.

Some remarks must be made concerning bindings and derived NF-properties. Let P be a derived NF-property and let $Q_1, ..., Q_n$ be the NF-properties 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-property bound to a component may be used as bound to an operation if the context requires it.

We now amplify the example presented in fig. 3. First of all, we declare all NF-properties in *CREATION_ISSUES* to be bound to components with a single declaration in the module header. On the other hand, NF-properties in *RELIABILITY* are not bound to any unit, so every NF-specification importing this module should state the binding (unless system modules do this job, see below). This is done in *LIBRARY*, which binds *fully_portable* and *reliability* to the component itself and *test* to all the operations introduced in the component (with the abbreviation "ops(LIBRARY)"). However, as far as *reliability* is a component-bound derived property, *test* must be bound also to components, so we have introduced a new property *test* at the component level, defined in terms of the operation-bound *test* (in this context, "min" is an operator returning the minimum value of a property in a set of operations). Also, we have bound *error_recovery* to the operations of *LIBRARY* and to *LIBRARY* itself; the component-bound property is defined as derived, in terms of the operation-bound ones. Last, we introduce a new efficiency property to measure the time required to execute a specific programming task performed with *LIBRARY*.

```
property module CREATION_ISSUES
    bound to components
    properties
        string responsible_name
        ...
end module

property module RELIABILITY
    imports CREATION_ISSUES
    properties
        boolean fully_portable
        ...
end module

non-functional specification module for LIBRARY
    imports RELIABILITY, CREATION_ISSUES
    binds fully_portable, reliability bound to LIBRARY
            test bound to ops(LIBRARY)
    properties
        integer test bound to LIBRARY derived
            depends on test(ops(LIBRARY))
            defined as test = min(test, ops(LIBRARY))
        boolean error_recovery bound to ops(LIBRARY)
        boolean error_recovery bound to LIBRARY derived
            depends on error_recovery(ops(LIBRARY))
            defined as
                    error_recovery = for all op in ops(LIBRARY) it holds error_recovery(op)
        efficiency time(list_books_borrowed_by_all_members) bound to LIBRARY
    measurement units n_books, n_members, n_book_copies, n_book_copies_borrowed
end module
```

Fig. 4: Declaration of non-functional properties with bindings for a LIBRARY component.

NF-properties will usually be applicable to all the components of a software system. With the kind of constructors presented up to now, this requires an explicit binding of these properties in every component, which is clearly inappropriate. So, we have introduced into NoFun a new kind of encapsulation mechanism called *system module*, to group the NF-properties that characterise a whole system.

A system module includes:

- The names of the software systems (note the plural usage) it is bound to.
- The names of the property modules to be imported in all the components of these systems.
- For all the NF-properties appearing in these property modules without any binding, some bindings may optionally appear. These bindings will be valid in all the components of the specified systems.
- Optionally, new NF-properties may be defined (with bindings or not). These NF-properties will be known in all the software systems listed in the header.

Fig. 5 shows a system module bound to two different software systems, which are supposed to use the *LIBRARY* component. The module imports the property modules *RELIABILITY* and *CREATION_ISSUES*, so the NF-specification of *LIBRARY* need not import them. Also, we have included the bindings for the *RELIABILITY* properties in the system module, and the *test* NF-property at component level. Note the "all operations" abbreviation, to refer to all the public operations introduced in the components appearing in *UPC_LIBRARY* and *LSI_LIBRARY*.

```
system module for UPC_LIBRARY, LSI_LIBRARY
    imports RELIABILITY, CREATION_ISSUES
    binds fully_portable, reliability bound to components
        test bound to all operations
    properties
        integer test bound to components derived
            depends on test(all operations)
            defined as test = min(test, all operations)
end module

non-functional specification module for LIBRARY
    properties
        boolean error_recovery ...
        boolean error_recovery derived ...
        efficiency time(list_books_borrowed_by_all_members) ...
    measurement units ...
end module
```

Fig. 5: A system module bound to two software systems and its effect in a component of them.

4 Statement of 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 *V* for a given software component *D* should state its NF-behaviour with respect to the basic NF-properties characterising *D*; values of derived NF-properties are automatically computed. This assignment of values is encapsulated in a NF-behaviour module.

For instance, the behaviour for an implementation *IMP_LIBRARY_1* for the *LIBRARY* component may look like the module in fig. 6. See the use of arithmetic operators for stating efficiency, interpreted in the big-Oh notation [Bra85]. Also, we use an "except for" construct with an obvious meaning. With this assignment, the value of the (component-bound) derived properties are: *external_responsible* = false, *test* = 2, *reliability* = *medium* and *error_recovery* = true.

```
behaviour module for IMP_LIBRARY_1
    behaviour  (* about CREATION_ISSUES *)
        responsible_name = "franch"; month = 3; year = 1997
                (* about RELIABILITY *)
        fully_portable; test(ops(LIBRARY)) = 4 except for test(check_out) = 2
                (* time and space of operations and type representation *)
        ...time(list_all_members) = n_members
        time(list_books_borrowed_by_a_member) = log(n_members) + n_book_copies_borrowed
                (* others *)
        error_recovery(ops(LIBRARY))
        time(list_books_borrowed_by_all_members) =
            time(list_all_members) + (n_members * time(list_books_borrowed_by_a_member))
end module
```

Fig. 6: Non-functional behaviour of an implementation for LIBRARY.

Some domains have default assignment values: false, for boolean properties; 1 (i.e. O(1) or constant complexity) for asymptotic properties; and the minimum value in the specified range for integer and real properties (or 0 if this range has not been declared).

5 Statement of 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 properties and to represent the environment where implementations are to be inserted.

NF-requirements may appear in both NF-specification and NF-behaviour modules.

5.1 Completing non-functional specifications

NF-requirements are used in non-functional specifications to build the non-functional part of the component specification, once the NF-properties characterising the component are known. More precisely, these NF-requirements state the conditions that every implementation of a software component must fulfil, and they may appear at three different places: system modules, in which case the NF-requirement applies in all the components of the systems listed in the header; property 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.

We add below a few NF-requirements in the NF-specification module for *LIBRARY*, including its imported property modules. Most of the requirements are up to the specifier (coming from a requirements analysis of the problem), but the last one in *LIBRARY* is really a universal property of the component to be fulfilled in all its implementations (the operator '>=' must be interpreted as set inclusion [Bra85]). Note that the implementation *IMP_LIBRARY_1* (see fig. 6) satisfies the NF-specification for *LIBRARY*.

```
system module for UPC_LIBRARY, LSI_LIBRARY
    imports RELIABILITY, CREATION_ISSUES
    binds and properties ... as before
    requirements fully_portable; test(all operations) >= 2 (* global requirements of the system *)
end module

property module CREATION_ISSUES
    properties ... as before
    requirements  year >=1990 (* date of creation of the company *)
end module

non-functional specification module for LIBRARY
    properties and measurement units ... as before
    requirements
        not external_responsible; not error_recovery => test < 5
        time(list_books_borrowed_by_all_members) >=
            time(list_all_members) + (n_members * time(list_books_borrowed_by_a_member))
end module
```

Fig. 7: Completing non-functional specifications with non-functional requirements.

Finally, note that there are implicit NF-requirements due to the existence of derived NF-properties. So, the requirements stated for *test*, *external_responsible* and *fully_portable* imply that the value of *reliability* must be *medium* or *high*. On the other hand, the NF-requirement for *external_responsible* implies that *responsible_name* must be one of "franch", "jones" or "smith".

5.2 Relating measurement units

NF-requirements are the main tool for building non-functional specifications, but there is still one point left. Note that 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 (we can think about mappings, with two measurement units, for keys and values); 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, such as "pot(n_members, 2) <= n_members * n_books", which is essential in order to find out if an implementation satisfies this kind of NF-requirements.

For instance, let *UPC_LIBRARY* be both a software component importing *LIBRARY* and the root of the hierarchy corresponding to the *UPC_LIBRARY* system. Then, relationships could be stated between the four measurement units introduced in *LIBRARY*; we could say that the number of books is the square (in the asymptotic world) of the number of members, that the number of copies of a book is negligible with respect to the number of books and members, and also that all the copies of a book may be checked out at the same time. The result appears in fig. 8, where '<<' means "negligible". Given these relationships and the model of the asymptotic properties domain, every boolean NF-expression involving those measurement units yields true or false, as is the case with "pot(n_members, 2) <= n_members * n_books", which equals true.

```
non-functional specification module for UPC_LIBRARY
    properties ...
    relationships
        n_books = pot(n_members, 2)
        n_book_copies << n_members
        n_book_copies_borrowed = n_book_copies
end module
```

Fig. 8: Non-functional specification for a component using LIBRARY.

Other kinds of relationship may be stated in the component introducing the measurement units. For instance, *LIBRARY* could include the relationship "n_book_copies_borrowed <= n_book_copies", which must obviously hold whatever the values of these units.

5.3 Fixing implementations for imported components

NF-requirements appearing in the NF-behaviour module bound to an implementation *V* state the conditions that the implementations of the software components imported by *V* must fulfil. 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.

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 *LIST* the following: first, implementation must be as reliable as possible; next, the cost of insertions and removals must be constant; last, traversal should be as fast as possible. Concerning *SET*, the particular implementation *SET_BY_HASHING* is directly selected.

```
behaviour module for IMP_LIBRARY_1
   behaviour ... as before
   requirements
      on LIST:
            max(reliability);   (* the enumerated property reliability has
                                     been declared ordered *)
            time(put, remove) = 1;
            min(time(traversal))
      on SET: implemented with SET_BY_HASHING
end module
```

Fig. 9: Non-functional requirements over lists in an implementation for LIBRARY.

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 [Sit92, Fra94] for differents proposals to avoid such restrictions).

5.4 Propagating NF-requirements through the hierarchy

Since a software system is a hierarchy of software components, NF-requirements in upper modules must somehow be taken into account in lower ones. For instance, let *UPC_LIBRARY* be defined as in 5.2 and importing both *LIBRARY* and *LIST* components, and let *UPC_LIBRARY_IMP* be an implementation for it stating the NF-requirements appearing in fig. 10 (*n_items* is a measurement unit introduced in *LIST*). The question that arises is: how to combine the NF-requirements on *LIST* stated in *IMP_LIBRARY_1* with the ones in *UPC_LIBRARY_IMP*?

```
non-functional behaviour module for UPC_LIBRARY_IMP
   behaviour ...
   requirements on LIBRARY: time(add_book) = 1; max(reliability)
                 on LIST: time(find) = log(n_items)
end module
```

Fig. 10: NF-requirements over imported components in an implementation for UPC_LIBRARY.

By default, we consider that inside *IMP_LIBRARY_1* NF-requirements for *LIST* stated in *UPC_LIBRARY_IMP* have more priority than those stated in *IMP_LIBRARY_1*; this is to say, the list of NF-requirements for *LIST* inside *UPC_LIBRARY_IMP* is implicitly added at the beginning of the list inside *IMP_LIBRARY_1*. As in many situations this may be inconvenient, NoFun offers two alternatives:

- The list of NF-requirements appearing inside *IMP_LIBRARY_1* can be declared as *absolute*. This means that propagation does not really take place; instead, the only requirements considered on *LIST* in *IMP_LIBRARY_1* are those appearing in its own list.
- The NF-requirements for *LIBRARY* appearing inside *UPC_LIBRARY_IMP* may include *subordinate* NF-requirements for *LIST*, which form a new, local context for propagation (see fig. 11). These subordinate NF-requirements could even contradict the global ones.

```
non-functional behaviour module for UPC_LIBRARY_IMP
   behaviour ...
   requirements
      on LIST: time(find) = log(n_items)
      on LIBRARY: time(add_book) = 1; max(reliability)
         where on LIST: fully_portable; time(find) <= n_items
end module
```

Fig. 11: Including subordinate NF-requirements in an implementation for UPC_LIBRARY.

6 Genericity and Inheritance

In this section, we sketch the treatment in NoFun of a pair of component structuring mechanisms: genericity and inheritance. In the case of genericity, we must also consider its dual mechanism, actualisation (that is, obtaining a concrete component from the generic one).

6.1 Genericity and actualisation of generic components

As in the functional counterpart, NF-specifications for generic components are parameterised by certain symbols, NF-properties in this case, which we refer to as *formal parameters*; these non-functional formal parameters are inferred from the functional ones, types and operations, which are the symbols that are really declared in the NF-specification. The NF-properties and measurement units bound to a generic component or to some operations of the component will exist for every further actualisation of this component, and their value in implementations will be determined by the NF-behaviour of the generic component together with the actual parameter passing performed in the actualisation; an actualisation will be valid only if there are one or more implementations satisfying the requirements with the given parameter passing.

For instance, we present in fig. 12 a generic non-functional specification for *LIST*. Besides declaring and binding some NF-properties and introducing the measurement unit *n_items* for the number of elements in the list it is stated that the type *elem* of elements and a comparison operation *eq* are formal parameters of this specification; these (functional) formal parameters give rise to three NF-properties that parameterise the NF-specification: space(*elem*), time(*eq*) and space(*eq*), the first two being related by a NF-requirement. The behaviour module associated with an implementation *ORDERED_LINKED_LIST* gives values to properties in the usual way, sometimes involving the formal parameters as happens in "time(find)" and "time(print_list)". When carrying out an actualisation of *LIST* in *LIBRARY*, we state the non-functional specification that defines the actual parameters bound to the formal ones, which will be the type of books and an operation to compare books (both defined in *BOOK_INFO*) in the case of creating lists to obtain the books checked out by a member; as a result, the NF-requirement appearing in *LIST* is satisfied by the *ORDERED_LINKED_LIST* implementation because its value for the NF-property "time(find)" equals to "log(n_elems) * space(book_info)" with this parameter passing.

```
non-functional specification module for LIST
    parameterised by
        type elem   (* gives rise to space(elem) *)
        ops eq      (* gives rise to time(eq) and space(eq) *)
        requirements  (* to be satisfied in valid actualisations *)
                time(eq) <= space(elem)
    imports and binds...
    properties ...
        boolean ordered bound to LIST
        efficiency time(print_list) bound to LIST
    measurement units n_elems
    requirements ...
end module

non-functional behaviour module for ORDERED_LINKED_LIST
    behaviour ...
        ...
        ordered; time(find) = log(n_elems) * time(eq)
        time(print_list) = n_elems * space(elem)
    requirements ...
end module

non-functional specification module for LIBRARY
    imports BOOK_INFO
                LIST[BOOK_INFO] where elem is book_info, eq is book_eq
    ...
    requirements on LIST: time(find) <= log(n_elems) * time(book_eq)
end module
```

Fig. 12: Generic non-functional modules for LIST and their use in LIBRARY.

There are some further issues concerning genericity that we do not mention here. For instance, with the constructs presented here some ambiguities could arise if there were more than one actualisation of *LIST* in *LIBRARY*. Also, we should take into account that, generally, a component implementation may have more parameters than its specification (for instance, a particular hash function in a set implementation); moreover, different implementations could have different parameters. Last, NoFun allows generic NF-specifications to be parameterised by other kind of properties apart from asymptotic ones.

6.2 Inheritance

In the case of inheritance, NF-specifications (including NF-properties with bindings and requirements, and measurement units with relationships) are inherited by derived components. Also, new NF-properties and measurement units may be added and older ones may be redefined. There are many correctness conditions that should be preserved to avoid inconsistencies and ambiguities.

7 Conclusions

A language to state non-functional specifications and requirements of software systems at the product level has been presented. The language complements work previously done at the process level [MCN92, CNY95, LS95] and may also be seen as an enrichment of classical specification languages (Z, Larch, OBJ3 and so on) which focus mainly on the statement of functionality of systems. 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 in the component programming framework; in fact, and trying not to be too ambitious, we would like this approach to be seen as a first step towards the definition of a non-functional notation to be widely adopted in this framework.

It is worth pointing out what we consider the salient features of NoFun:

- We think that it is concise, clear and easy to use. For instance, NoFun expressions look like normal expressions in a high-level programming language, using a few *ad hoc* operators to cover some specific non-functional concepts, mainly concerning efficiency. Obviously, due to the complexity of the subject, NoFun is not trivial in appearance.
- NoFun specifications are a constituent part of software. This requirement highlights the role of non-functional issues in software development and it is achieved in many ways: NoFun descriptions are encapsulated in modules bound to software component definitions and implementations, and the notation is entirely formal in the sense that its syntax and semantics are well-defined [Fra96].
- It does not depend on the actual (functional) specification and programming languages used to develop the system. The only restriction we put on the programming language is modularity: software components must be really encapsulated 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.
- NoFun is not restricted to a particular software area. Although we have focused here on component programming, we think that NoFun fits other areas well with little effort, and this is a current line of research in our work. Of course, some modifications cannot be avoided: for instance, the measurement strategy of efficiency we present in the paper could not be applied to information systems.

Also, it is worth mentioning that NoFun is complemented by an algorithm to select the best implementation of a component in every context where it is used. This is very important to support software maintenance and reusability, and also quality of design. The existence of this algorithm allows the whole software system to adapt automatically to changes in the environment, once the descriptions made with NoFun are modified conveniently [FB97].

In addition to the selection algorithm, some aspects of our work have not been explained here. For instance, neither the problem of interaction of data implemented in different ways nor the possibility of stating NF-requirements in particular places of programs (such as variables or type components) have been addressed; also, the formal definition of NoFun (mathematical models for domains of NF-properties, correctness conditions, consistency, satisfactibility of requirements, etc.) has not

been included. All of these features play an important part in our proposal; details may be found at [Fra94, FB96, Fra96].

Last, there is an important point we have still not addressed: automatic synthesis of values of NF-properties in implementations. This is to say, we have provided no means in our proposal to compute the value of a specific basic NF-property from the code of the implementation; we are only able to calculate the value of derived NF-properties from the corresponding formula. Note that if full automatic synthesis were carried out, NF-behaviour modules could disappear. Automatic synthesis is a current line of research in our team, and we are starting to apply abstract interpretation to compute program efficiency. However, it must remain clear that there are many NF-properties whose values do not seem to be easily computable from code; an example is the *test* property used in this paper. In any case, we think that the NoFun notation is useful even without automatic synthesis, just as a specification language.

As far as we know, there is no proposal for a language with the constructs presented in this report. 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-properties in an object-oriented framework; no other kind of properties are dealt with in her approach. An interesting approach 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 measure 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 notations used in all of these approaches, however, are not as powerful as NoFun, even considering just the subset of NoFun concerning efficiency.

References

[Bra85] G. Brassard. "Crusade for a Better Notation". SIGACT News, 16(4), 1985.

[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.

[CNY95] L. Chung, B.A. Nixon, E. Yu. "Using Non-Functional Requirements to Systematically Support Change". In *Procs. of 2nd Symposium on Requirements Engineering*, York (England), 1995.

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

[CZ91] S. Cárdenas, M.V. Zelkowitz. "A Management Tool for Evaluation of Software Designs". IEEE Transactions on Software Engineering, 17(9), 1991.

[FB96] X. Franch, X. Burgués. "Incremental Component Programming with Functional and Non-Functional Information". In *Proceedings of XVI International Conference of Chilean Computing Science Society*, Valdivia (Chile), 1996.

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

[Fra94] X. Franch. "Combining Different Implementations of Types in a Program". In *Proceedings Joint of Modular Languages Conference*, 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.

[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 ESEC*, Barcelona (Catalunya, Spain), 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 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 *Procs. 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*, 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 CAiSE*, Utrecht (Holland), LNCS 811, 1994.

[Win89] J.M. Wing. "Specifying Avalon Objects in Larch". In *Proceedings of TAPSOFT'89, Vol. 2*, Barcelona (Catalunya, Spain), LNCS 352, 1989.

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

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Research Reports – 1997

LSI–97–1–R "On the Number of Descendants and Ascendants in Random Search Trees", Conrado Martínez and Helmut Prodinger.

LSI–97–2–R "On the Epipolar Geometry and Stereo Vision", Blanca García de Diego.

LSI–97–3–R "Solving Incidence and Tangency Constraints in 2D", Núria Mata.

LSI–97–4–R "*Designer:* A Tool to Design and Model Workflows", Camilo Ocampo, Pere Botella.

LSI–97–5–R "OBJECTFLOW: A Modular Workflow Management System", Camilo Ocampo, Pere Botella.

LSI–97–6–R "The Extreme Vertices Model (EVM) for Orthogonal Polyhedra", A. Aguilera and D. Ayala.

LSI–97–7–R "An Improved Master Theorem for Divide-and-Conquer Recurrences", Salvador Roura.

LSI–97–8–R "Randomized Binary Search Trees", Conrado Martínez and Salvador Roura.

LSI–97–9–R "Programming Frames for the Efficient Use of Parallel Systems", Thomas Römke and Jordi Petit i Silvestre.

LSI–97–10–R "Concurrent Rebalancing of AVL Trees: A Fine-Grained Approach", Luc Bougé, Joaquim Gabarró, Xavier Messeguer, and Nicolas Schabanel.

LSI–97–11–R "The VEX-93 Environment as a Hybrid Tool for Developing Knowledge Systems with Different Problem Solving Techniques", Julio J. Valdés, Ramón Hita, Katia Peón, Dania Hernández, Ada García, Raul Paredes, Yazna García, and Alfredo Rodríguez.

LSI–97–12–R "Single-Pushout Hypergraph Rewriting through Free Completions", Ricardo Alberich, Francesc Rosselló, and Gabriel Valiente.

LSI–97–13–R "Design, implementation and evaluation of ParaDict, a data parallel library for dictionaries", Joaquim Gabarró and Jordi Petit i Silvestre.

LSI–97–14–R "NoFun: A Notation to State Non-Functional Specifications at the Product Level", Xavier Franch.

Hardcopies of reports can be ordered from:

Nuria Sánchez
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo, 5
08028 Barcelona, Spain
secrelsi@lsi.upc.es

See also the Department WWW pages, http://www-lsi.upc.es/www/