Supporting Software Maintenance with Non-Functional Information

Xavier Franch, Pere Botella
franch@lsi.upc.es, botella@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 highlights the role of non-functional information (about efficiency, reliability and other software attributes) of software components in software maintenance, focusing in the component programming framework. Non-functional information is encapsulated in modules bound to both definitions and implementations of software components and it is written as expressions in a classical programming language. It is shown with an example how this notation supports software maintenance, with the help of an algorithm which is able to select the best implementation of a software component in its context of use, meaning by "best" the one that better fits to its non-functional requirements. As a conclusion, we may say that our proposal will probably reduce maintenance costs in case of software modifications due to changes in the non-functional environment of the system and also to changes in the NF-behaviour of software components, including migration to other platforms.

1. Introduction

Software systems may be seen as the composition of many software components which work together to accomplish their goals. They are characterised both by their functionality (i.e., what the system does) and by their non-functionality (i.e., how 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 and, in particular, there exist just a few proposals of formally-defined notations to express non-functional information of software systems.

The absence of explicit statement of non-functional issues have a negative impact on many aspects of the software process, including software maintenance. There are many common scenarios which would benefit from this kind of non-functional information appearing in software systems:

- The environment of the system changes with respect to its expected non-functional behaviour. We mention here: variations on response time requirements, moving from the existing platform to another one, etc.
- A software component in the system is modified in a way such that its non-functional characteristics vary. This could happen, for instance, when developing the system as a sequence of prototypes: new versions of software components may improve execution time, or reliability through exhaustive testing, etc.
- A new version of a software component is built with a different non-functional behaviour compared to other existing versions. A typical situation would be a software factory producing a new version of a reusable component.

Note that all of these situations require studying the non-functional behaviour of (part of) the software system and eventually they also require the ability to compare two functional-equivalent software components with respect to their non-functional behaviour and/or to test if a software component satisfies some non-functional constraints. So, the existence of non-functional information in the software system itself would improve the achievement of these tasks reducing thus the high cost of maintaining and evolving existing software.

In this paper, we present a notation (formally defined in [7]) to support the statement of non-functional information and we study its feasibility in software maintenance through a small (for the sake of brevity) example. Up to now, our research has addressed to the *component programming* framework as defined in [8, 12], which is characterised by the existence of components representing abstract data types, with: 1) a *definition* stating the type's name and its operations; 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.

0-8186-7892-5/97 $10.00

Non-functional information of software components is actually encapsulated in *ad hoc* modules bound to them and it is classified into three kinds: declaration of non-functional properties, statement of non-functional behaviour and statement non-functional requirements. The notation we propose has been designed with the goal of simplicity in mind (to improve software development and understanding) and it is complete enough to allow automatic selection of the "best" version of a software component, meaning by "best" the one that better fits to the non-functional requirements of the component in the system. This automatic selection has a positive impact on the whole software development process and, in particular, on software maintenance. Also, it should be said that our proposal does not depend on the underlying programming language used to code the system, provided that component definitions and implementations can be defined in independent modules (as it is the case, for instance, in the O.-O. family).

2. Stating Non-Functional Information on Software

We classify non-functional information into three different kinds:

- *Non-functional property* (short, *NF-property*). Any attribute of software which serves as a mean to describe it and possibly to evaluate it. For instance, time and space efficiency, reusability degree and reliability. Up to now, we have not defined any predefined catalogue of NF-properties, but the mechanisms to define them.

  In the general case, we study a given software component 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*). Evaluation of the set of NF-properties which characterise this component.
- *Non-functional requirement* on a component implementation (short, *NF-requirement*). Any constraint on its NF-behaviour referred to one or more NF-properties from the set characterising the component.

2.1. Declaration of NF-properties

NF-properties are declared in *NF-declaration modules*, bound to software component definitions. They may be declared directly or else they may be imported from *property modules*, which define sets of related NF-properties that can be used in different systems and which allow users to define their own catalogues of NF-properties.

NF-properties may be of four different types, depending on the domain of their values:

- Boolean. NF-properties that simply hold or fail, as full portability.
- Numerical. NF-properties which can be measured somehow, as reusability degree.
- By enumeration. NF-properties which can be classified into some categories, as user interface (icons, command language, etc.).
- Efficiency. To establish the execution time and space of exported operations and types.

  Efficiency properties need not to be explicitly declared (they come into existence from the corresponding software component definition); instead, it is necessary to provide a set of *measure units* to modelise input data sizes that can have influence on efficiency.

In the component programming framework, efficiency can be measured with a class of functions using the *big-Oh asymptotical notation* [2], used to establish efficiency of programs for a great amount of input data and defined as:

$$O(f) = \{g: \mathcal{N}^+ \to \mathcal{N}^+ \,/\, \exists c_0 \in \mathcal{N}^+,\ \exists n_0 \in \mathcal{N}^+: \forall n \geq n_0: g(n) \leq c_0 f(n)\}$$

$\mathcal{N}^+$ stands for positive integers and f is a function characterising the efficiency of a type or operation. The definition has been given for a single measure unit; it can be extended for an arbitrary number of them.

We present next an example. The software component *NETWORK* (fig. 1) represents geographical distributions of items with a connection cost (money, distance, time, etc.); both items and connection costs are represented by positive integer numbers. There are operations for adding and removing items and links, to obtain the shortest path connecting two items and to simplify the network in such a way that all the items are connected and the global cost is as low as possible (i.e., to compute a minimum spanning tree for the network). Lists of items include connection costs; so, a component defining lists of pairs of integers is imported.

```
definition module NETWORK
  imports LIST_OF_PAIR_OF_INTEGERS
  type network
  operations
    create returns network
    add_item, rem_item (network, int) returns network
    add_link (network, int, int, int) returns network
    rem_link (network, int, int) returns network
    shortest_path (network, int, int)
                        returns list_of_pair_of_int
    simplify_network (network) returns network
end module
```

Fig. 1: The NETWORK software component definition.

Concerning non-functionality, we have chosen as NF-properties: three boolean ones to tell if the component is fully portable, if it has been coded by an external programmer and if the implementation uses the dynamic storage mechanism of the language; a property by enumeration to determine which kind of data structure is used in the type; and two numerical properties to state the reliability of the component and the number of links of the representation. We have chosen to define some of them in a separate property module, so that they can be used in many other different systems. Note the coexistence of very abstract NF-properties (e.g., reliability) with more concrete ones (e.g., number of links). Concerning efficiency, we have introduced two measure units, one for the number of items and the other for the number of connections.

```
declaration module for NETWORK
   imports IMPL_ISSUES
   properties
      boolean fully_portable, external_programmer
      numerical confidence_correctness [0..5]
         (* 0 -> 5: increasing degree of testing *)
   measure units n_items, n_conns
end module
```

```
property module IMPL_ISSUES
  properties
    boolean dynamic_storage
    numerical nb_links
    enumerated data_structure
          = (hashing, avl, heap, chained, others, none)

end module
```

Fig. 2: NF-properties for the NETWORK software component.

2.2. Statement of NF-behaviour

Each software component implementation *V* for a given software component definition *D* should state its NF-behaviour with respect to *D*'s NF-properties in a *NF-behaviour module*, bound to *V*. To be more precise, each implementation should define: which logical NF-properties hold, the value for every numerical and enumerated NF-property, and the time and space costs for the exported types and procedures.

So, the cost for an implementation *IMP_NET1* for the definition *NETWORK* using adjacency lists, an improved Dijkstra algorithm (with heaps) to find out the shortest path and the Kruskal algorithm for the minimum spanning tree computation may look as in fig. 3. Note the use of arithmetic operators in stating efficiency, interpreted in the big-Oh notation as explained in [2].

```
behaviour module for IMP_NET_1
  behaviour
    fully_portable; confidence_correctness = 3
    dynamic_storage; nb_links = 1
        data_structure = hashing
    space(network) = n_items + n_conns
    time(create) = n_items
    time(add_item, rem_item) = n_items
    time(add_link, rem_link) = n_items
    time(shortest_path) =
                (n_items+n_conns)*log(n_items)
    time(simplify_network) = n_conns*log(n_items)
    space(shortest_path, simplify_network) =
                n_items + n_conns
end module
```

Fig. 3: Behaviour for an implementation of NETWORK.

By default, auxiliary space for procedures is O(1) and logical properties do not hold.

2.3. Statement of NF-requirements

NF-requirements state conditions on implementations of software components. Syntactically, they are boolean expressions enriched with some *ad hoc* constructs for non-functionality (see examples below). Their purpose is to represent the environment where implementations are to be put in. They may appear both in NF-declaration and NF-behaviour modules and they may involve again measure units introduced in NF-declaration modules.

NF-requirements in NF-declaration modules state the conditions that an implementation must fulfil in order to be useful in the software system. They affect both the development of new implementations and the reuse of existing ones. We modify in fig. 4 the declaration module for *NETWORK* including some relationships between NF-properties and measure units.

```
declaration module for NETWORK
  imports IMPL_ISSUES
  properties
    boolean fully_portable, external_programmer
    numerical confidence_correctness [0..5]
  measure units n_items, n_conns
  relations
    n_conns <= pot(n_items, 2)
    dynamic_storage => confidence_correctness < 5
    (not fully_portable and external_programmer) =>
        confidence_correctness = 0
end module
```

Fig. 4: The NETWORK software component definition including NF-requirements.

NF-requirements in NF-behaviour modules state the conditions that an implementation must fulfil in order to be used in another implementation. NF-requirements should appear for every software component imported by an implementation and they should be complete enough to select a single implementation for each of these components; also, a concrete implementation for a software component may be selected directly by its name.

In NF-behaviour modules, it is also possible to state a list of NF-requirements over a definition, which are applied in order of appearance; this corresponds with the usual case of having requirements with different degree of importance. NF-requirements in the list are applied until one of the following three conditions holds:

- A single implementation is selected.
- Applying the next NF-requirement would yield an empty set of implementations.
- All the NF-requirements have been applied

In the last two cases, more than one implementation may satisfy a given list and then requirements would have to be reviewed (in fact, the algorithm may be tuned so that a single implementation is selected from the set of candidates satisfying the list of NF-requirements).

For instance, a NF-requirement over lists in the *IMP_NET_1* implementation could be: "time efficiency of individual operations and their auxiliary space must be negligible (i.e., O(1)); next, list traversal should be as fast as possible; last, and in order of importance, implementation must be reliable, fully portable and with the fewest links per cell". These requirements can be expressed as in fig. 5.

```
behaviour module for IMP_NET_1
  behaviour ...
  requirements
    on LIST_OF_PAIR_OF_INTEGERS:
      time(put, delete, get) = 1 and
        space(ops(LIST_OF_PAIR_OF_INTEGERS)) = 1;
      min(time(traversal));
      max(confidence_correctness);
      fully_portable; min(nb_links)
end module
```

Fig. 5: NF-requirements over lists in an implementation of NETWORK.

3. Support to Software Maintenance

In this section, we study how the notation presented here (with the help of the implementation selection algorithm described in [7]) supports software maintenance. First of all, we introduce two more implementations for *NETWORK*, *IMP_NET_2* and *IMP_NET_3*, with the following NF-behaviour (we just show the properties used in the example).

```
behaviour module for IMP_NET_2
  behaviour
    fully_portable; external_programmer
    confidence_correctness = 5
    time(add_item, rem_item) = 1
    time(add_link, rem_link) = 1
    time(shortest_path) = pot(n_items, 2)
    time(simplify_network) = pot(n_items, 2)
  requirements ...the same as IMP_NET_1
end module
behaviour module for IMP_NET_3
  behaviour
    not fully_portable; not external_programmer
    confidence_correctness = 3
    time(add_item, rem_item) = n_items
    time(add_link, rem_link) = n_items
    time(shortest_path) = pot(n_items, 2)
    time(simplify_network) = pot(n_items, 2)
  requirements ...the same as IMP_NET_1
end module
```

Fig. 6: NF-behaviour for two more implementations of NETWORK.

Next, we define the initial context for the *NETWORK* component. Let the context be a software system for a national railway network which main goal is to find out shortest paths between pairs of train stations. We represent the system with a software component definition *RAILWAY*, implemented with a module *RAILWAY_IMPL*. According to the railway environment, we state two kind of NF-requirements:

- In *RAILWAY*, we define *n_conns* to be asymptotically equal to *n_items*, as it is the case in a usual railway network. Note that this assignment satisfies the relation *n_conns <= pot(n_items, 2)* stated in *NETWORK*, as it is necessary to happen.

```
declaration module for RAILWAY
  ...
  relations n_conns = n_items
end module
```

Fig. 7: Adding NF-information in RAILWAY.

- In *RAILWAY_IMPL*, we constrain *NETWORK* in the following way. First, we require an implementation with a good response time for *shortest_path*; second, in case of more than one implementation satisfying this main goal, we require them to be confident enough and to be made by a non external programmer; last, we require the implementation to be fully portable.

```
behaviour module for RAILWAY_IMPL
  behaviour ...
  requirements on NETWORK:
      time(shortest_path) <= pot(n_items, 2);
      confidence_correctness >= 3 and not
                external_programmer;
      fully_portable
end module
```

Fig. 8: Adding NF-requirements over NETWORK in an implementation of RAILWAY.

The evaluation of these requirements on the existing implementations yields to the following result:

time(shortest_path) <= pot(n_items, 2): satisfied by all implementations (even by *IMPL_NET1*, because of the equality *n_conns = n_items*)

confidence_correctness>=3 and not external_programmer: satisfied by *IMPL_NET1* and *IMPL_NET3*; so, *IMPL_NET2* is discarded

fully_portable: satisfied by *IMPL_NET1* and not by *IMPL_NET3*

The implementation selection algorithm chooses then *IMPL_NET1* to be used in the software system for the railway network.

3.1. Changes on the Current Platform

Let's suppose that, after a few crashes in the railway software system, the railway company decides to give preference to software reliability over other considerations. This decision is easily taken into account just by changing the NF-behaviour module for *RAILWAY_IMPL* (see fig. 9).

```
behaviour module for RAILWAY_IMPL
  behaviour ...
  requirements on NETWORK:
    max(confidence_correctness);
    time(shortest_path) <= pot(n_items, 2);
    not external_programmer; fully_portable
end module
```

Fig. 9: Changing NF-requirements over NETWORK.

Then, the implementation selection algorithm should be executed again, selecting *IMPL_NET_2* as the new implementation for *NETWORK*.

3.2. Moving to a new Platform

Now, let's suppose that we reuse the *NETWORK* component in a software system for a wide area computer network with a nearly fully-connected topological configuration. In this network, nodes may temporally disconnect when their local work load is too high; also, connection costs may vary depending on some factors. So, operations for adding and removing nodes and links are executed very often and they must be optimised. Every time the network configuration changes, we require to execute the minimum spanning tree algorithm to have all the nodes connected with the minimum global cost; so, this operation must be optimised too.

The change of platform with these non-functional information is represented with the modules shown in fig. 10.

```
declaration module for COMPUTER_NETWORK
  ...
  relations n_conns = pot(n_items, 2)
                (* nearly fully-connected network *)
end module
```

```
behaviour module for COMPUTER_NETWORK_IMPL
  behaviour ...
  requirements on NETWORK:
    time(add_item, rem_item, add_link, rem_link) = 1
                and min(time(simplify_network))
end module
```

Fig. 10: NF-information appearing in a new platform using NETWORK.

This NF-information leads to *IMPL_NET_2* as the implementation automatically selected for *NETWORK*.

3.3. Renovating Software Components

Once the software system is operative, some variations may occur in its non-functional behaviour as time goes by. Some of them may result from improvements of the components; for instance, when a component has been more carefully tested or when its efficiency is improved somehow; modifications made when the component becomes an aging piece of software also fall in this category. Other changes may arise from the environment evolution, as it happens if the component's programmer moves from the software company. Anyway, such a change requires just modifying the non-functional modules bound to the component and then re-running the implementation selection algorithm.

3.4. Creating new Implementations for Software Components

Other situation we think our approach is well-suited to deal with is the creation of new versions for software components. Note that this case is similar to the former one: it is necessary to create the NF-modules for them and then re-running the implementation selection algorithm in all the software systems where the component is used, because it may be the case that this new implementation

fits in some contexts better than the previously chosen ones.

4. Conclusions

A notation for stating non-functional issues of software systems in the component programming framework has been presented. We have shown its usefulness in supporting software maintenance due to changes in non-functional characteristics of the environment or the software itself. The notation is complete enough to express NF-properties of software components in a way such that the best implementations for them in every context where they appear are automatically selected depending on their NF-behaviour; contexts are represented by means of NF-requirements (a list of boolean expressions). NF properties may be boolean, numerical, by enumeration of values or concerning efficiency. Implementations should state both their NF-behaviour with respect to their corresponding properties, and also their NF-requirements on every used component.

Our proposal provides many interesting features. First, programmers just establish NF-requirements and NF-behaviour of software components; implementations are automatically selected avoiding then a bad design (with respect to selection of implementations). Second, software is robust with respect to changes on NF-requirements and construction of new implementations, requiring just re-running the implementation selection algorithm. Last, information about non-functionality is a constituent part of software, improving thus its understanding and making easier the communication between designers, implementers and users of components; all of these features support software maintenance, especially the second one. On the other hand, there seems to be no drawbacks in our approach, because the NF-language is conceptually simple, with a syntax resembling classical expressions and it is not bound to any particular programming language provided that definitions and implementations are kept separated.

Some aspects of our work have not been explained here because they are out of scope of the paper. For instance, neither the working procedure of the automatic selection algorithm (for which there is a prototype) nor the problem of interaction of data implemented in different ways have been studied. They play an important part in our system but they do not directly affect software maintenance. For details, see [6, 7].

Our approach has currently some limitations. There is no way to verify that a software component implementation really exhibits the stated NF-behaviour. In fact, we are interested not in verifying but in extracting NF-behaviour from implementations whenever possible. For instance, we are studying the application of abstract interpretation techniques to compute automatically efficiency of operations and types, as done in [1]. A related problem is to choose which metrics do we use to measure the most usual NF-properties others than efficiency. Also, we are starting to adapt our proposal from component programming to the information systems field, which demands some changes on the kind of NF-information managed (for instance, asymptotic efficiency has to be replaced for time measured in fractions of seconds), although the main ideas are the same. Last, we want to complement our product-oriented approach by a process-oriented one, yielding thus to a software process in which non-functionality plays a crucial role (as done in [11]).

As far as we know, there exists no proposal for a language with the constructs presented in this report, although there have been many claims in this sense [8, 13, 17]. There are many non-formalised proposals [9, 10] which results are subsumed in our work. Also, [16] presents an interesting case study to deal with boolean NF-properties into an O.-O. framework; no other kind of properties are dealt in her approach. On the other hand, [3] and [15] offer the possibility to select implementations from some efficiency information appearing in programs; however, the constructs they offer are not as powerful as ours. [14] makes also a proposal oriented to software reuse but limited to efficiency again.

The approach closest to ours is [4, 5], which provides a framework to evaluate the design of software systems, the measure 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 result in a number and comparison is possible. The proposal is not integrated in the software itself and then the selection of implementations is not automatic.

References

[1] Y. Aït-Ameur. "Formal Program Development by Transformation and Non Functional Properties Evaluation". In *Proceedings of the 5th SEKE International Conference*, IEEE, 1993.

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

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

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

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

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

[7] X. Franch. "Automatic Implementation Selection for Software Components using a Multiparadigm Language to state Non-Functional Issues". Ph.D. Thesis, Universitat Politècnica de Catalunya, (Spain), 1996.

[8] M. Jazayeri. "Component Programming - a Fresh Look at Software Components". In *Proceedings of 5th ESEC*, Barcelona (Catalunya, Spain), 1995.

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

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

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

[12] "Special Feature: Component-Based Software Using RESOLVE". ACM Software Engineering Notes, 19(4), 1994.

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

[14] M. Sitaraman. "On Tight Performance Specification of Object-Oriented Components". In *Proceedings of the 3rd International Conference on Software Reuse*, IEEE, 1994.

[15] P.C-Y. Sheu, S. Yoo. "A Knowledge-Based Program Transformation System". In *Proceedings 6th CAiSE*, Utrecht (Holanda), LNCS 811, 1994.

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

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