Browsing a Component Library using Non-Functional Information†

Xavier Franch[1], Josep Pinyol[1], and Joan Vancells[2]

[1] Universitat Politècnica de Catalunya (UPC),
c/ Jordi Girona 1-3 (Campus Nord, C6) E-08034 Barcelona (Catalunya, Spain)
franch@lsi.upc.es, josep.pinol.molas@arthurandersen.com
[2] Universitat de Vic (Barcelona),
c/Sagrada Família, 7 E-08500 Vic (Catalunya, Spain)
joan.vancells@uvic.es

Abstract. This paper highlights the role of non-functional information when reusing from a component library. We describe a method for selecting appropriate implementations of Ada packages taking non-functional constraints into account; these constraints model the context of reuse. Constraints take the form of queries using an interface description language called NoFun, which is also used to state non-functional information in Ada packages; query results are trees of implementations, following the import relationships between components. We define two different situations when reusing components, depending whether we take the library being searched as closed or extendible. The resulting tree of implementations can be manipulated by the user to solve ambiguities, to state default behaviours, and by the like. As part of the proposal, we face the problem of computing from code the non-functional information that determines the selection process.

1 Introduction

Software components can be characterised both by their functionality (what the component does) and by their non-functionality (how the component behaves with respect to some quality factors like efficiency, reliability, etc.). Both aspects should be considered during their specification, design, implementation, maintenance and also reuse. If we focus on reusability, a component retrieved from a library regarding only its functional behaviour may not fit into the non-functional requirements of the environment, hindering or even preventing its actual integration into the new system.

Despite this, usual software reuse methods (see [1] for a survey) take only functional characteristics of components into account. The main reason behind this limitation is that non-functional information does not appear in components; furthermore, it often cannot be easily computed (or even cannot be computed at all) from the func-

† This work has been partially supported by the spanish CICYT project TIC97-1158.

tional part. As a result, retrieval of components cannot guaranty success with respect to non-functional constraints (for instance, constraints about efficiency).

Our approach cope with this problem by coupling three different strategies. On the one hand, we have designed an interface definition language called *NoFun* aimed at stating non-functional aspects of software components in the components themselves. This notation allows to introduce software attributes characterising components, libraries and whole systems; to put constraints on them; and also to establish how component implementations behave with respect to them. On the other hand, we have defined a method for retrieving software components from a library, based on the satisfactibility of some non-functional constraints encapsulated in queries. The method examines component implementations from a non-functional point of view; it will be defined in two different scenarios depending on whether we require to solve the query using just the implementations of the library or not. Last, a framework for computing non-functional information of implementations from code has also been formulated. The framework seems to be able to handle a relevant subset of quality factors (among them, we remark efficiency) in a uniform way.

The paper focuses on the last two parts (especially the second one) and not on the language, which have been formerly described in [2] and later in [3]. A first prototype running on Linux is currently implemented.

2 The Framework

In the rest of the paper, we will view a software component as a pair built up from an *interface* and an *implementation*. In our approach, the interface (enclosed in an Ada package) optionally includes an Anna *specification* (see [2] for details). In the general case, there will be components with the same specification but different implementations (enclosed in Ada package bodies), each one designed to fit a particular context of use. Our reuse method will be on charge of selecting one or more functional-equivalent components with the most appropriate implementation for a given context represented by a query.

On the other hand, components are considered encapsulations of abstract data types. Implementations will consist then of data structures as lists, trees and graphs. This framework affects the kind of relevant non-functional attributes of components; for instance, we measure efficiency with asymptotic notations [4] and not by response time, throughput or number of accesses to disk, which are measures of interest when considering other type of systems.

Non-functional information of software components will be actually integrated into Ada packages by means the NoFun interface description language (lines starting with "--|"). We classify this information into three kinds (see [3] for details):

– *Non-functional attribute* (short, *NF-attribute*): definition of software attributes which serve as a mean to describe components and possibly to evaluate them. Among the most widely accepted we can mention efficiency, maintainability, reliability and usability.

– *Non-functional behaviour* of an implementation (short, *NF-behaviour*): assignment of values to the NF-attributes bound to the component.

– *Non-functional constraint* on a software component (short, *NF-constraint*): constraint on the set of the NF-attributes bound to the component.

In fig. 1 we show an example of definition of an NF-attribute for reliability, which relies on three other NF-attributes: test degree (integer from 0 to 5), error recovery and portability (both boolean).

```
package RELIABILITY is
   --: with ERROR_RECOVERY, TEST
   --: properties
   --:  boolean FullyPortable;    -- platform independence
   --:  enumerated ordered Reliability [none,low,medium,high]
   --:      depends on Test, ErrorRecovery, FullyPortable
   --:      defined  as
   --:        not ErrorRecovery and not FullyPortable =>
   --:                    Reliability = none
   --:        ErrorRecovery and not FullyPortable =>
   --:                    Reliability = low
   --:        not ErrorRecovery and FullyPortable =>
   --:                    Reliability = low
   --:        ErrorRecovery and FullyPortable =>
   --:                    Test in [0..1] => Reliability = low
   --:                    Test in [2..3] => Reliability = medium
   --:                    Test in [4..5] => Reliability = high
end RELIABILITY;
```

Fig. 1. A package introducing a NF-attribute for reliability

This attribute can be used in component packages, as shown in fig. 2; *NETWORK* is a component modeling geographical networks, which establishes connections between items. We assume that items in the network are integers, and that the cost of a connection is also an integer. The *measurement units* stand for data volume sizes: *NbItems* for the number of items, and *NbConns* for the number of links. These units are used later to establish efficiency results.

```
package NETWORK is
   ... declaration of interface
   --: with RELIABILITY
   --:    measurement units NbItems, NbConns
end NETWORK;
```

Fig. 2. A package for a component using the NF-attribute of fig. 1

Fig. 3 includes a NF-behaviour module giving values to these NF-attributes (*ShortestPath* is a procedure computing shortest paths in the network). Note that the value of the *Reliability* NF-attribute is not explicitly given, because it can be computed from the other ones (we say it is *derived*). The *implemented with* construct labels the package body for further package selection.

```
package body NETWORK is
   --| implemented with ADJACENCY_MATRIX
   --| behaviour
   --|      ErrorRecovery; FullyPortable; Test = 3
   --|        -- this implies Reliability = Medium
   --|      space(Network) = pow(NbItems, 2)
   ...
   procedure ShortestPath ...
   --|      time(ShortestPath) = pow(NbItems, 2)
   --|      space(ShortestPath) = NbItems
   ...
end NETWORK;
```

Fig. 3. A package implementation for the component in fig. 2, including NF-behaviour

3 Queries

In our method, a query is the basic retrieval operation. It is aimed at selecting components whose implementation fit better in the new system from a non-functional point of view, identifying which conditions must hold in order for this selection to be correct. Query process relies on the NF-behaviour of implementations; this is why in the rest of this section we talk about implementations instead of components. For every NF-attribute in the scope of the implementation, the NF-behaviour states its value, either implicitly or not, depending on whether the attribute is basic or derived; the NF-constraints appearing in the behaviour module are used to fix the additional conditions. One of the most important basic attributes is efficiency, whose can be computed with the help of a tool (see section 7); we plan to extend the set of basic NF-attributes computed in this way.

In order to combine queries later on, we assume that the selection is made from an initial set of candidate implementations; so, queries may be viewed as mappings that bind sets of implementations. More precisely, given a query q_M and a set S of implementations for an interface M (such that for every $s \in S$, the pair $<M, s>$ is a component from our point of view), the evaluation of q_M over S, written $q_M(S)$, yields:

- A set $T \subseteq S$ such that implementations in T satisfy the conditions appearing in q_M.
- A T-indexed family $Q_T = (q_t)_{t \in T}$, such that q_t is in turn a V-indexed family of queries, $q_t = (q_{t,v}(R_v))_{v \in V}$, being V the set of components imported by t and R_v the set

of all the implementations of the component v. The query $q_{t,v}(R_v)$ represent the conditions that the imported implementation v must fulfil in order for t to be selected by the original query.

We call the queries in Q_T *subordinated queries* and then q_M becomes the *main query*. When referring to the result of the evaluation of q_M, eval(q_M), we will write as eval(q_M).T the set T and by eval(q_M).Q_T the family of queries Q_T.

From the syntactic point of view, a query is defined as a list of *atomic queries*, $q_M(S) = aq_{M,1}(S_1) \oplus ... \oplus aq_{M,k}(S_k)$. Items in the list represent conditions in decreasing order of importance; so, the initial set S can be restricted little by little until obtaining the final result. Atomic queries are very close to NF-constraints, except that we require them appearing in conjunctive normal form (CNF) and that we can use a pair of useful operators to select implementations maximizing or minimizing the value of a given NF-attribute. The operator $\oplus$ is called *restriction operator*, and it is defined in 3.2.

3.1 Atomic Queries

We define an atomic query $aq_M(S)$ as an expression given in CNF, possibly negated, $aq_M(S) = A$ or $aq_M(S) = \neg A$, such that $A = A_1 \wedge ... \wedge A_r$; de Morgan laws are applied to eliminate disjunctions, and so A_i may become also negated, $A_i = B_i$ or $A_i = \neg B_i$. Logical connectives are in fact interpreted as set operators. Every B_i can be:

- A relational expression, comparing expressions of a measurable attribute domain. Given the input set S, the evaluation of the relational expression $E_1 < E_2$ (for any defined ordering <), denoted by eval($E_1 < E_2$), is defined as ($E[R]$ stands for the evaluation of E with the values appearing in the behaviour module bound to R):

$$\text{eval}(E_1 < E_2) = \{ R \in S \,/\, E_1[R] < E_2[R] \}.$$

- A quantification of the form *max* or *min*, to select a subset of implementations inside S maximizing or minimizing a given expression E, defined as:

$$\text{eval}(\max(E)) = \{ R \in S \,/\, (\forall T \in S\text{: } E[T] \leq E[R]) \}.$$

$$\text{eval}(\min(E)) = \{ R \in S \,/\, (\forall T \in S\text{: } E[T] \geq E[R]) \}.$$

The evaluation eval($aq_M(S)$) of the atomic query $aq_M(S)$ is as follows:

- Computation of T requires the evaluation eval(B_i) of all B_i, which results in sets $S_i \subseteq S$. If $A_i = B_i$, then evaluation eval(A_i) of A_i equals S_i; if $A_i = \neg B_i$, it equals $S - S_i$. Then, we define the evaluation of A as eval(A) = eval(A_1) $\cap$ ... $\cap$ eval(A_r). Finally, we define eval($aq_M(S)$) = eval(A) if $aq_M(S) = A$, and also eval($aq_M(S)$) = S - eval(A) if $aq_M(S) = \neg A$.

- The T-indexed family $Q_T = (q_t)_{t \in R}$ results in a V-indexed family of queries, $q_t = (q_{t,v}(R_v))_{v \in V}$, such that V is the set of all imported components in t, R_v is the set of all the implementations of v, and $q_{t,v}$ is the NF-constraint stated on v inside t, which will be assumed to be *true* if no such NF-constraint exists.

3.2 Combination of Atomic Queries

We define here the meaning of the restriction operator $\oplus$ that combines atomic queries to give the result of the main query. In fact, we give two different definitions considering two cases. In the first case, we focus on obtaining the best implementations for the component of interest, even if there are not implementations for the imported components in the library; we call it *open case*. In the second one, the *closed case*, the restriction operator assumes that the implementations in the library are enough to satisfy not only the main query but also the subordinated ones.

The Open Case. The main idea behind open query computation is to evaluate atomic queries in order of appearance, until obtaining a single implementation for the abstract data type being reused and thus the component is uniquely defined; the result of an atomic query is considered as the input of the following one. However, there are two cases that do not fit into this scheme:

- Even after processing all atomic queries, more than one implementation is still possible. In this case, all of them are considered as the result of the query.
- An atomic query is not satisfied by any of the implementations resulting from the previous one. In this case, we consider as the result of the query the implementations obtained in this previous atomic query.

In both situations, some user interaction is required for selecting one of them (see section 6).

The evaluation is defined in two steps. First, we define the connection between two consecutive atomic queries by connecting their input and output sets:

$$S_1 = S.$$
$$S_i = \text{eval}(aq_{M,i-1}(S_{i-1})).\text{T},\ 1 < i \leq k.$$

Now, the evaluation of the query $q_M(S)$ is stated as:

$$\text{eval}(q_M(S)) = \text{eval}(aq_{M,i}(S_i)),\ 1 \leq i \leq n, \text{ such that:}$$
$$|\ \text{eval}(aq_{M,i}(S_i)).\text{T}\ | = 1 \wedge\ (i > 1 \Rightarrow |\ \text{eval}(aq_{M,i-1}(S_{i-1})).\text{T}\ | > 1$$
$$\vee$$
$$|\ \text{eval}(aq_{M,i}(S_i))\ | > 1 \wedge (i < k \Rightarrow |\ \text{eval}(aq_{M,i+1}(S_{i+1})).\text{T}\ | = 0).$$

The computation of the family of queries is straightforward from the set.

As a correctness condition for evaluation of queries, it must hold that $\text{eval}(aq_{M,1}(S_1)).\text{T} \neq \emptyset$.

The Closed Case. If we choose to obtain a result such that all the queries are solved with the existing implementations, it could be the case that restricting excessively the set of implementations in a query processing leads to unsolvable subordinated queries. So, we redefine the evaluation of query $q_M(S)$ preventing this case. The definition uses a predicate *solvable* that checks if there is a unsolvable subordinate query; as subordinated queries may activate others, the predicate takes a recursive form:

$$\text{eval}(q_M(S)) = \text{eval}(aq_{M,i}(S_i)),\ 1 \le i \le n, \text{ such that:}$$
$$\forall q\text{: } q \in \text{eval}(aq_{M,i}(S_i)).\text{Q}_\text{T}\text{: solvable(q)}$$
$$\wedge$$
$$\{\ \mid \text{eval}(aq_{M,i}(S_i)).\text{T} \mid = 1 \wedge (i > 1 \Rightarrow \mid \text{eval}(aq_{M,i-1}(S_{i-1})).\text{T} \mid > 1$$
$$\vee$$
$$\mid \text{eval}(aq_{M,i}(S_i)).\text{T} \mid > 1 \wedge (i < k \Rightarrow \mid \text{eval}(aq_{M,i+1}(S_{i+1})).\text{T} \mid = 0 \vee$$
$$\exists q\text{: } q \in \text{eval}(aq_{M,i+1}(S_{i+1})).\text{Q}_\text{T}\text{: } \neg\text{solvable(q)}\ \}.$$

Being:

$$\text{solvable}(q) \equiv \text{eval}(q).\text{T} \neq \emptyset \wedge \forall q'\text{: } q' \in \text{eval}(q).\text{Q}_\text{T}\text{: solvable}(q').$$

As a correctness condition for the evaluation of queries, it must hold both that $\text{eval}(aq_{M,1}(S_1)).\text{T} \neq \emptyset$ and $\forall q$: $q \in \text{eval}(aq_{M,1}(S_1)).\text{Q}_\text{T}$: solvable($q$).

4 Selection Trees

As the evaluation of queries is defined in a recursive form, it is natural to use trees of implementations to represent its result; we call them *selection trees*. One could think also to use directed graphs, but it would be incorrect since an implementation selected in two different queries may use different implementations for one or more of its imported components.

Selection trees consist of the following elements:

- *Nodes*. We distinguish two types of nodes: *interface nodes*, represented by ellipses, and *implementation nodes*, represented by rectangles. There is a special implementation node, called *void*, that appears when there are not implementations satisfying a particular query.
- *Branches*. There are two types of branches: *import branches*, going from implementation nodes to interface ones, and represented by arrows; and *selection branches*, going from interfaces to implementations and represented by undirected lines.

In fact, component nodes are not strictly necessary, but we include them for clarity reasons and also to support some kinds of user interaction.

5 An Example

Let *NETWORK_USER_IMPL* be a package using *NETWORK* as defined in figure 2. Let's assume that this new component is mainly devoted to compute shortest paths in the network by means of the operation *ShortestPath*. Also, let's assume that the network is nearly fully connected.

There exist different implementations for *NETWORK*, which differ in two points: the strategy to represent the underlying graph (we focus in adjacency lists and adja-

cency matrix) and the algorithm that implements the operation *ShortestPath*. With respect to the first point, an additional fact must be considered: how the data structure is indexed using items (integers). Note that we have three different cases: the items are known in advance; the items are not known but the number of items is bounded; and there is no information about the items. Let's assume the second case.

Under this assumption, implementations will use instances of a generic *MAPPING* component to access the data structure via items. In the case of an adjacency list, the mapping associates lists to items; in the adjacency matrix case, it returns integers between 1 and *N* to access the matrix.

Operations on mappings are insertion, deletion and retrieval using the item as a key. So, standard mapping implementations will be useful here: hashing, AVL trees, ordered lists, etc. In the case of hashing, chained strategy will use in turn the generic *LIST* component to link synonymous keys.

Lastly, note each of these implementations will also use the *LIST* component, to build adjacency lists when necessary and to build the result of *ShortestPath*.

To keep the example in a reasonable size, we restrict the set of implementations of the components introduced so far (see fig. 4): three implementations for lists (ordered, unordered with pointers and unordered with arrays), two for mappings (chained hashing and AVL trees) and also two for priority queues (heaps and AVL trees with access to minimum element), which are used in some versions of the *ShortestPath* implementation (improved Dijkstra algorithm).

We are going to process the query below, which can be read as: first, minimizing the time of finding shortest paths with a reasonable reliability (the reliability can be relaxed if we are just building a first prototype of the application); next, minimizing type representation space; last, maximizing reliability. As initial set for the query, we take the set of all *NETWORK* implementations introduced above. All of this, with the (asymptotic) relationship *NbConns* = power(*NbItems*, 2) (being *NbConns* and *NbItems* the representation in NoFun sentences of the number of connections and items, necessary to state efficiency), coming from the fully connected characteristic of the network (in the query, the semicolon stands for the composition operator):

```
min(time(ShortestPath)) and Reliability >= medium  ;
min(space(Network))  ;   max(Reliability)
```

Fig. 5, left, shows the selection tree of the query; its generation follows from the non-functional characteristics of the components. Two ambiguities arise which correspond to the existence of two pairs of siblings. To make them more visible, we draw an ambiguity sphere linking all the selection branches that stem from the same node. Explicit interaction with the user is needed, either to provide an additional query to solve every ambiguity, or to choose an implementation directly by name. In this case, *UNORDERED_POINTERS*, which is valid in both contexts, seems to be preferable.

```
------------ Implementations for NETWORK
--  All of them require:
--  from MAPPING, fast accessing time;
--  on lists, the first two ones dynamic storage
--       (to avoid wasting space), and all of them
--        require fast insertion time for building the
--        result of ShortestPath.

ADJ_LISTS_1: -- adjacency lists and Dijkstra algorithm
                -- with priority queues.
  space(Network) = NbItems + NbConns
  time(ShortestPath) = (NbItems+NbConns) * log(NbItems)
  Reliability = medium
  requires on MAPPING: min(time(Insert, Delete, Get))
           on LIST: min(time(Put)) and DynamicStorage
           on PRIORITY_QUEUE:
                  time(Put, First, RemFirst) <= log(NbElems)

ADJ_LISTS_2: -- adjacency lists and Dijkstra algorithm
  space(Network) = NbItems + NbConns
  time(ShortestPath) = power(NbItems, 2)
  Reliability = medium
  requires on MAPPING: min(time(Insert, Delete, Get))
           on LIST: min(time(Put)) and DynamicStorage

ADJ_MATRIX: -- adjacency matrix and Dijkstra algorithm
  space(Network) = power(NbItems, 2)
  time(ShortestPath) = power(NbItems, 2)
  Reliability = high
  requires on MAPPING: min(time(Insert, Delete, Get))
            on LIST: min(time(Put))


------------ Implementations for LIST

ORDERED: -- keep elements in order; makes use of pointers.
   time(Put) = NbElems; DynamicStorage

UNORDERED_POINTERS: -- keep elements without order;
                       -- makes use of pointers.
   time(Put) = 1; DynamicStorage

UNORDERED_ARRAY: -- elements stored in an array, linking
                    -- them, without any order.
   time(Put) = 1; not DynamicStorage


------------ Implementations for HASHING

CHAINED_HASHING: -- hash table linking synonymous with lists
  time(Insert, Delete, Get) = 1; not DynamicStorage
  require on LIST: DynamicStorage    -- number of collisions
                                        -- not known in advance

AVL:          -- an AVL tree making use of dynamic storage.
  time(Insert, Delete, Get) = log(NbElems); DynamicStorage


------------ Implementations for PRIORITY QUEUE

HEAP: -- elements stored in an array managed as a heap
  time(Put, First, RemFirst) = log(NbElems)
  not DynamicStorage

AVL_WITH_MIN: -- an AVL with an additional pointer to the
                 -- minimum element.
  time(Put, First, RemFirst) = log(NbElems)
  DynamicStorage
```

Fig. 4. Non-functional behaviour (highlights) for some components

To illustrate the importance that relationships between different efficiency parameters have during selection, we reformulate the same query in a different situation, considering networks with a few connections. This situation can be modeled with the (asymptotic) relationship *NbConns* = *NbItems*, and then the query selects as network implementation the first one, because Dijkstra algorithm takes profit of the use of priority queues. Now, implementation for lists is uniquely determined due to the additional constraint of using dynamic storage for them, while implementation for mappings do not vary. Concerning priority queues, both existing implementations satisfy the NF-constraints, and so both appear in the resulting selection tree (fig. 5, right).

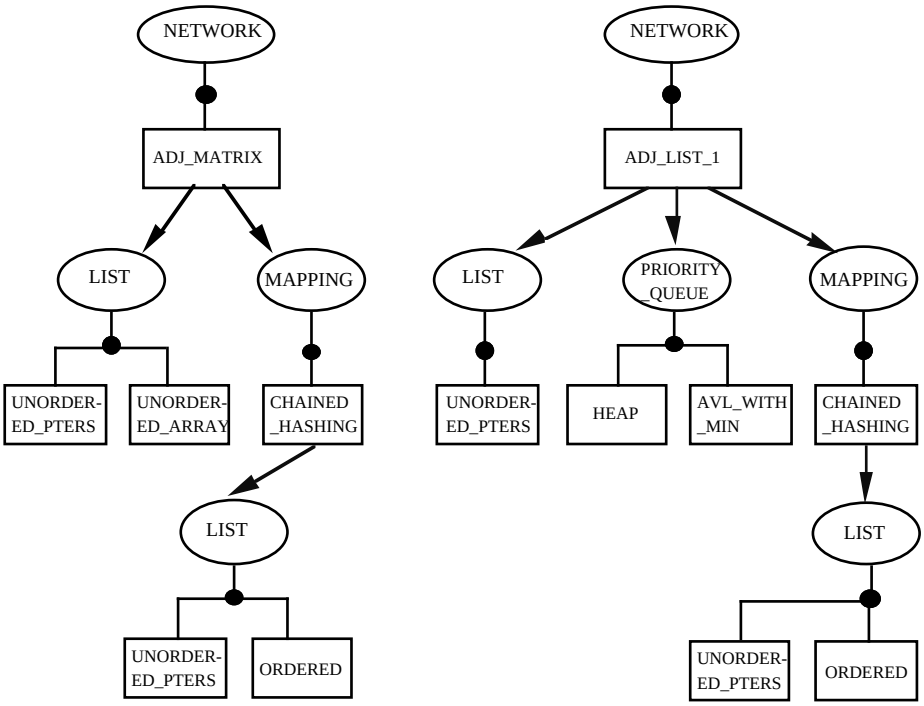

Fig. 5. Two selection trees for a query

6 User Interaction

In order to make our proposal more useful, the method that we have presented is complemented with the ability to guide the process and eventually affect its result. There are many reasons supporting this decision:

– As seen in the example, query evaluation may yield ambiguous selection trees. Users should then choose one of the existing alternatives, or else an additional query may be formulated on the involved component.

- Some users may prefer not to fully rely on the query processing algorithm but only to guide the process and processing atomic queries little by little, perhaps stopping the process before than expected or switching from one type of query to another to compare results.
- Correctness conditions may be violated either in the main query or in subordinate ones. In the first case, the query should be relaxed and the process started again. We do not allow selection of implementations violating queries.

To carry out this interaction, the selection tree is depicted in the screen as seen in fig 6. Ambiguities could be solved either by clicking one of the candidate implementations, or else by clicking the ambiguity sphere and typing a query in a new window. This last action will be also possible when clicking a void node, to edit the unsolved query and make it less restrictive.

7 Automatic Computation of Efficiency

Efficiency is computed putting together three different elements: patterns, a synthesized attribute in the grammar, and program annotations. First of all, a synthesized attribute has been defined for computing efficiency from the syntactical layout of the code. This solves some cases, but obviously this attribute is not able to handle computation with loops.

Loops can be annotated. This means that the programmer can include NoFun expressions in the Ada code which state the cost of the loop, probably in terms of the efficiency parameters.

Obviously, annotations are uncomfortable to deal with; they require an extra effort from programmers, and also prevents our scheme to be applicable to existing programs. So, we have defined patterns as the mean to identify program schemes which determine some efficiency results. Patterns can be global, this is to say, independent of the component, as the ones for computing efficiency of recursive procedures and loops incrementing a control variable; or they can be specific, when some program schemes on components are identified, as for instance patterns for graph traversals. This last kind of patterns are of interest not only to avoid annotations, but also to obtain more accurate results.

Once asymptotic expressions are obtained, we can apply a simplification calculus to obtain reduced expressions, which are the final results to be assigned to the NF-behaviour of components.

8 Conclusions

A proposal for software reuse that takes non-functional issues into account has been presented. The proposal consists of a language to state non-functional attributes, be-

haviour and constraints; a method for reuse which takes the form of queries written with the same notation; and a framework for computing some quality factors, presently efficiency. The result of the query is a tree of implementations, modifiable by the user, which follows the import relationships between components. Reuse has been studied in two different situations, depending on whether we force the resulting tree to be completed using just the implementations of the library or not. The efficiency of the selection method is not a function on size of the library, but on: 1) the average number of component implementations, 2) on the average height of selection trees, and 3) the amount of atomic queries that appear in non-atomic ones; these factors will generally be moderate enough to assure a good response time.

The most related approach to ours is the faceted-classification scheme presented in [5] and which is the basis of other proposals, many of them implemented on the WWW [6, 7]. In [5], facets are mostly used to select components by the functionality their provide; however, the ideas can be applied to non-functionality by considering facets as enumerated NF-attributes, which are a particular domain constructor in No-Fun. In this sense, our approach generalizes facets by allowing other kind of domains. Another difference is that [5] incorporates a notion of similarity between components; we have not adopted this idea because the language presents other features that allow to rank components (ordered domains) and to retrieve those ones maximizing / minimizing the value of one or more NF-attributes. Lastly, [5] does not allow to retrieve trees of components but just individual ones. On the other hand, there is an interesting feature which we do not provide at the moment, namely the fuzziness quality of some NF-attributes.

References

1. Mili, H.; Mili, F.; Mili, A.: Reusing Software: Issues and Research Directions. IEEE Transactions on Software Engineering 21, 6. IEEE Computer Society (1995)
2. Franch, X.: Including Non-Functional Issues in Anna/Ada Programs for Automatic Implementation Selection. Proceedings of Ada Europe 97 (London, UK). Lecture Notes in Computer Science, Vol. 1251. Springer-Verlag, Berlin Heidelberg New York (1997)
3. Franch, X.: Systematic Formulation of Non-Functional Characteristics of Software. Proceedings of International Conference on Requirements Engineering (ICRE) (Colorado Springs, USA). IEEE Computer Society (1997)
4. Brassard, G.: Crusade for a better Notation. SIGACT News, 16, 4 (1985)
5. Prieto-Díaz, R.: Classifying Software for Reusability. IEEE Software 4, 1. IEEE Computer Society (1987)
6. Boisvert, R.F.: A Web Gateway to a Virtual Mathematical Software Repository. Proceedings of 2nd International WWW Conference, Chicago (Illinois, U.S.A.) (1994)
7. Poulin, J.S.; Werkman, K.J.: Melding Structured Abstracts and the WWW for Retrieval of Reusable Components. Proceedings of Symposium on Software Reusability (Seattle, U.S.A.) (1995)