Using Non-Functional Requirements in Component-Based Software Construction[1]

Pere Botella, Xavier Franch, Xavier Burgués
botella@lsi.upc.es, franch@lsi.upc.es, diafebus@lsi.upc.es
Dept. Llenguatges i Sistemes Informàtics (LSI)
Technical University of Catalunya (UPC)
Pau Gargallo 5, 08028 Barcelona, Catalunya (Spain)
FAX: 34-3-4017014. Phone: 34-3-4016994

Abstract

The main concern of this paper is to present the author's approach to support software development in the component programming framework taking functional and non-functional requirements into account. Functional requirements are written as algebraic specifications, while non-functional information is bound to specifications and implementations by means of *ad hoc* modules: the non-functional information is used to select automatically the most appropriate implementations of software components (the selection algorithm is not presented here). The existence of multiple type implementations is supported by a process model based on the prototyping paradigm. Prototyping is achieved by means of a mixed execution mechanism being able to operate in the context of incremental software development process allowing the execution of incomplete (partially implemented) systems. The ideas we present here are not bound to any particular programming language, giving rise to a method of wide applicability.

Key words and phrases: component software design, non-functional requirements, prototyping.

1 Introduction

Component programming [RES94, Jaz95] is a useful and widely spread way of building complex software systems by means of combining, reusing and producing software components, and can be seen as a general term to denote well-known concepts as object-oriented or modular software development. What a component does is stated by its functional properties. Different implementations must satisfy them but they will differ on some non-functional aspects, as execution time or reliability.

Among other possibilities, we are interested in software components as an encapsulation of *abstract data types* (*ADT*), described by algebraic specifications and implemented using an imperative programming language. To be more precise, components consist of: a) the *definition* of an ADT stating both functional and non-functional characteristics, and b) one or more *implementations*, each one including a description of its non-functional behaviour. In this framework, we conceive the development of a system as a refining process, in which each step can be: Specify a new ADT; Add more properties to an incompletely specified operation; Start or continue an implementation of an ADT, possibly importing some other ADTs; Reuse a specific implementation of an ADT to fit some particular requirements on it; Execute the current system to test it for errors and/or to tune it up.

There are some aspects of our approach that makes them different (and, of course, better in our view) with respect to other related existing ones approaches: the definition of a complete and formal notation to state not only the usual functional aspects of software but also non-functional ones; the automatic selection of the implementation for a given specification, improving the reuse and maintaining multiple implementations, in a single system; an *ad hoc* process model addressed to

1 This work has been partially supported by the spanish research programme PRONTIC under contract TIC92-0667 and also by the OBJECTFLOW project.

support prototyping in the framework defined by the existence of multiple implementations; finally, execution is possible at each step of development, combining in the general case specifications and implementations (maybe different ones for the same ADT).

The rest of the paper is structured as follows. Section 2 presents the notation to write functional specifications and non-functional information (the implementation selection algorithm is not presented here, see [Fra96] and [FB96]). Section 3 describes the software process assistant. Section 4 presents the mixed execution mechanism. Finally, section 5 outlines some conclusions and mentions some related work.

2 A Notation to Express Functional and Non-Functional Information

We present in this section a formal notation to enrich an imperative programming language with both functional and non-functional information. The functional part is just outlined because it consists of well-known constructs in the algebraic specification framework. A more detailed explanation is given for the non-functional part, which we will call *NF-language* from now on.

2.1 Specification of functional behaviour

We adopt algebraic specifications as the language to state the functional behaviour of components. At least, a specification declares the name of the type being defined and the list of operations of the type, including their arity. Next, conditional equations for the type may appear, which are interpreted by default with initial semantics; optionally, operations may be left incompletely specified, stating just some universal equations that they must fulfill.

For instance, we present next a *NETWORK* software component specification, to be used as an ADT for activity networks, establishing precedence relationships between some kind of tasks (represented by natural numbers). We introduce operations to create an empty network, to create a network with an initial task (*create2*), to add and remove precedence relationships, to get the immediate successors of a task and to obtain a topological sort, which returns a valid order of execution of the tasks. The specification imports *LIST_OF_NATURAL* which is assumed to offer as many operations as necessary. We give the equations for *succ* and *top_sort*, that show the two usual kinds of specifications: *succ* is completely defined (i.e., its behaviour is uniquely determined for every network), while *top_sort* is not: it is just stated that if a task *m* precedes a task *n* then *m* must appear before *n* in the resulting list.

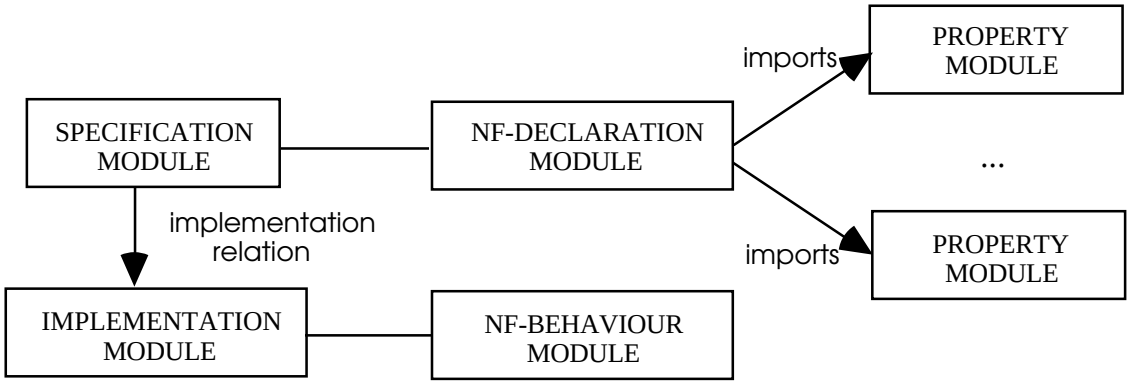
```
specification module NETWORK
imports LIST_OF_NATURAL
type network
operations   create returns network
             create2 (natural) returns network
             add, remove (network, natural, natural) returns network
             succ (network natural) returns list_of_natural
incomplete top_sort (network) returns list_of_natural
equations
    ... equations for the type (including error conditions)
    succ(create, m) = LIST_OF_NATURAL.empty
    succ(add(d, m, n), m) = LIST_OF_NATURAL.put(succ(remove(d, m, n), m), n)
    [m <> x] => succ(add(d, m, n), x) = succ(d, x)
    [belongs(succ(d, m), n)] => before(top_sort(d), m, n) = true
end module
```

2.2 The NF-language

The NF-language is designed to state three kinds of non-functional information:

- *Non-functional property* (short, *NF-property*): any attribute of software which serves as a mean to describe it and possibly to evaluate it. Among the most widely accepted ones [IEEE92, ISO91], we mention: time and space efficiency, reusability, maintainability, reliability and usability. In our approach, we let also more specific attributes to appear.
  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 *characterized* by this set.
- *Non-functional behaviour* of a component implementation (short, *NF-behaviour*): any assignment to the NF-properties characterizing the implemented software component.
- *Non-functional requirement* over a software component (short, *NF-requirement*): any constraint referred to some NF-properties from the set characterizing the component.

Next figure shows how non-functional information is collected into various modules. NF-properties are declared in *NF-declaration modules*, bound to specifications. They may import one or more *property modules*, which should be used to define NF-properties of wide applicability, appearing in many NF-declaration 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[2]. NF-behaviour of implementations is stated in *NF-behaviour modules*, bound to them. Also, NF-behaviour modules will usually include NF-requirements over the software components imported by the implementation.



Declaration of NF-properties

NF-properties appearing in NF-declaration modules may be of four different kinds: 1) boolean, to represent software attributes which simply hold or fail (ex.: full portability); 2) numerical, to introduce software attributes that can be measured (ex.: reusability degree); 3) by enumeration, to represent software attributes which can be classified into some categories (ex.: kind of user interface); 4) efficiency, to establish the execution time and space of types and operations. Lower and upper limits of numerical NF-properties may be specified; also, the set of valid values of a NF-property by enumeration may be listed together with the NF-property name.

Efficiency NF-properties need not to be explicitly declared; they come into existence from the corresponding software component specification. In the ADT framework, we measure efficiency with the *big-Oh asymptotic notation* [Knu76], which gives rise to a few specific operators in the NF-language, such as power, logarithm and by the like. Values of this kind of NF-properties are given in terms of some *measure units*, which represent problem domains' size and which must also appear in NF-declaration modules.

---

2 A current line of research is the development of a predefined catalogue containing the most widely accepted non-functional properties together with various existing metrics.

A NF-declaration module for *NETWORK* is shown next. Note that NF-properties at different abstraction levels (general ones as reliability and more concrete ones as programmer's name) can coexist freely. We have chosen to define a related subset of the NF-properties in a separate property module, *IMPLEMENTATION_ISSUES*, so that they can be used in other components. Concerning efficiency, we have introduced two measure units, one for the number of tasks and the other for the number of explicit precedence relationships.

```
property module IMPLEMENTATION_ISSUES
properties
boolean dynamic_storage
numerical links_number
enumerated data_structure = (hashing, avl, heap, chained, others)
end module
```

```
declaration module for NETWORK
imports IMPLEMENTATION_ISSUES
properties
boolean fully_portable, external_programmer
numerical reliability [0..5]
   (* 0: non-reliable; 1-4: increasing degree of testing; 5: formal verification *)
enumerated programmer_name
measure units n_tasks, n_preceds
end module
```

Statement of NF-behaviour

Each implementation *V* for a given software component specification *D* should state its NF-behaviour with respect to the NF-properties characterizing *D* in the NF-behaviour module bound to *V*. For instance, the behaviour of an implementation *IMP_NET_1* for the definition *NETWORK* using a graph implemented by an adjacency matrix may look as the module below. See the use of arithmetic operators to state efficiency interpreted in the big-Oh notation [Bra85].

```
behaviour module for IMP_NET_1
behaviour
    fully_portable; programmer_name = Franch; reliability = 3
    links_number = 0; data_structure = others
    space(network) = pot(n_tasks, 2); time(create, create2) = pot(n_tasks, 2)
    time(add, remove) = 1; time(succ) = n_tasks
    time(top_sort) = pot(n_tasks, 2); space(top_sort) = n_tasks
end module
```

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

Statement of NF-requirements

NF-requirements state conditions over implementations of software components. Syntactically, they are usual 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 modules and NF-behaviour ones and they may involve again measure units introduced in NF-declaration modules.

NF-requirements in NF-declaration modules state the conditions that every implementation of the component must fulfill in order to be useful in the system. We enrich below the declaration module for *NETWORK* with some relationships between NF-properties and measure units.

```
declaration module for NETWORK
imports IMPLEMENTATION_ISSUES
properties ... the same as before
measure units n_tasks, n_preceds
relations
   n_preceds <= pot(n_tasks, 2); dynamic_storage => reliability < 5
   (not fully_portable and external_programmer) => reliability = 0
end module
```

```
property module IMPLEMENTATION_ISSUES
properties ... the same as before
relations dynamic_storage => links_number > 0
   data_structure = heap => (links_number = 0) and not dynamic_storage
end module
```

NF-requirements appearing in a NF-behaviour module bound to an implementation V state the conditions that the implementations of all the software components imported by V must fulfill in order to be useful in V. They should be complete enough to select a single implementation for each imported software component. In the general case, V may include a list of NF-requirements over every imported component; NF-requirements in the list are applied in order of appearance[3] until one of the following conditions holds: 1) a single implementation is selected; 2) applying the next NF-requirement would yield to an empty set of implementations; 3) 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[4]. As an alternative, a concrete implementation for a component may be selected directly by its name.

For instance, a NF-requirement over *LIST_OF_NATURAL* in implementation *IMP_NET_1* could be: first, implementation must be as reliable as possible; next, the cost of the operations to build a list and their auxiliary space must be as fast as possible (i.e., O(1) in the big-Oh notation); last, traversal should be as fast as possible. We use a few predefined operators ˝which have an intuitive meaning.

```
behaviour module for IMP_NET_1
behaviour ... the same as before
requirements on LIST_OF_NATURAL:
   max(reliability)
   time(empty, put) = 1 and space(ops(LIST_OF_NATURAL)) = 1
   min(time(traversal))
end module
```

3 The Software Process Assistant

We describe in this section a software process assistant based on the idea of prototyping with incomplete programs. “Incomplete programs” stands for programs which may contain non-implemented types, and/or objects with no associated implementation. Execution of incomplete programs is interesting in the context of program development through prototyping because the choice

3 This corresponds with the usual case of having requirements with different degree of importance.

4 In fact, the algorithm may be tuned so that a single implementation is automatically selected from the set of candidates satisfying the list of NF-requirements.

of implementation for types and objects may be delayed until the entire behaviour of the module hierarchy has been proved correct. This section proposes to perform this “proof” through testing, by means of the construction of a sequence of executable prototypes.

Let *H* be a hierarchy of modules and let *lib* be a library of software components. The main capabilities offered by the assistant are:

1.- Define the signature of a module: **define(H, spec_mod)**. During this definition, more concrete orders may be used: **create(name)** to create the empty module, **add_type(spec_mod, name)** and **add_op(spec_mod, op)**; also, reuse from library can be made as explained at point 4.

2.- Specify a module: **specify(H, spec_mod)**. The module can be already defined or not. In order to obtain the equations, the assistant guides the user into the initial semantics framework, asking for:

2.i Which are the constructors of every type, that is, the minimal set of operations to build any value: **constructors(spec_mod, set_of_ops)**. Constructors may be introduced as private.

2.ii Which are the relationships among those constructors: **congruence(spec_mod, set_of_eqns)**.

2.iii For each other operation, which are the equations defining its behaviour: **specify_op(spec_mod, set_of_eqns)**.

In the last step, a default procedure may be applied under request to decompose the parameters of the operation being specified using the constructors of their type. Prototyping may be done at any point during specification; also, specification may be left (temporarily or permanently) incomplete, which may affect prototyping.

3.- Implement a defined or specified module: **implement(H, name_spec, impl_mod)**. In order to allow prototyping of the implementation to occur as soon as possible, the assistant asks for:

3.i The representation of the type plus its abstraction function plus the implementation of the constructor operations: **represent(impl_mod, name_type, repr)**, **abstr(impl_mod, name_type, set_of_eqns** or **code)** for the abstraction function (which may be specified or implemented, see next section) and **impl_op(impl_mod, name_op, code)** for each constructor.

3.ii For each other operation, its implementation: **impl_op(impl_mod, name_op, code)**.

So, prototyping may take place after 3.i and also after every operation implemented in 3.ii. Implementation may be left temporarily incomplete, which may affect prototyping.

4.- Reuse a module from the library: **use(mod, lib, name_spec, rens)** and **instantiate(mod, lib, name_spec, assoc, rens)**. That is, the module is used or instantiated inside a module *mod* that is being defined, specified or implemented; *rens* stands for the symbol renamings made and *assocs* stands for the binding between formal and actual parameters in the instantiation (the fitting morphism in the initial semantics approach). In case of using or instantiating a module with a given implementation, its name is added as the last parameter.

5.- Choose an implementation for an module, a type or an object: **choose_mod(mod, name_spec, name_impl)** for modules, **choose_type(mod, name_type, name_impl)** for types and **choose_const(mod, name_const, name_impl)**, **choose_funct_result(mod, name_funct, name_impl)** and **choose_var(mod, name_funct, name_var, name_impl)** for the appropriate

kind of object (the last one includes parameters); for modules, we may also use the orders in point 4. All the bindings are made inside the implementation module *mod*.

6.- Store a module into the library: **store(H, lib, module)**.

7.- Prototype a function: **prototype(H, name_module, name_function)**. The assistant asks for the input parameter values and then the required function is executed, calling either a conventional interpreter if the function is implemented or the term-rewriting system if not. Input parameter values may be persistent objects as well as input / output devices, which are treated as abstract data types. Prototyping is the crucial functionality of the assistant, and it can be invoked at any moment when specifying or implementing a module.

4 Mixed execution

We describe now the execution strategy that we have designed for our system. The main goal of this strategy is the ability to execute software systems at any stage of development, supporting thus a prototyping software process where implementations of components can be incrementally added and tested. It should be said that this process will usually be inefficient, but we are interested in testability rather than in efficiency; we remark that the final product will have all of its components totally implemented and it will not be executed using this strategy.

There are two situations that must be coped in order to assure this full executability. First, it can be the case for an ADT to be selected with more than a single implementation in the system. Second, the current version of the system may combine modules just specified with others for which an implementation has been selected; even more, specifications can contain operations not completely defined, while implementations may be only partial during their development.

4.1 Coexistence of multiple implementations

The coexistence of multiple implementations is problematic only when two objects of the same type but different implementations interact, as it happens in *concat*(*x*, *y*) and *x* := *y*, being *x* and *y* two lists implemented in different ways. To support this free interaction it is necessary to have a method to transform the corresponding data structures in the adequate manner. Instead of having particular methods for every pair of existing implementations, we have preferred to define a common framework where implementations are translated to and from: the specification of the type, and so the methods are: the *abstraction function* to transform a data structure to an expression (a *term*) formed by the operations of the type, and the *representation relation* to perform the inverse step; both constructions were already introduced in [Hoa72].

While the representation relation is no more than the execution of the operations appearing in the term into the goal implementation, the abstraction function must be explicitly given. It is coded as a special function returning a term and it appears inside NF-behaviour modules. We show below the abstraction function for the adjacency-matrix implementation *IMP_NET_1*:

```
behaviour module for IMP_NET_1
 ...
abstraction (n: network) returns t: TERM(network)
var i, j: integer
   t := create
  for i := 1 to max do    (* max: maximum number of tasks *)
  for j := 1 to max do if n[i, j] then t := add(t, i, j)
end module
```

Note the use of a predefined type constructor, *TERM*(*t*), used to modelise terms in the algebraic framework. Inside the abstraction function, calls to operations of the type being abstracted (*create* and *add* in the example) are interpreted as building-term operations.

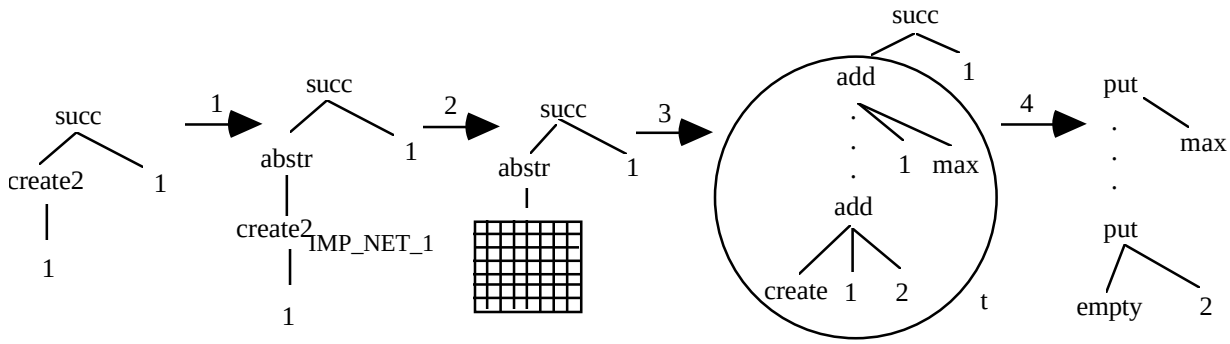
4.2 Coexistence of specifications and implementations

Incorporating specifications into the execution process is conceptually simple if we consider specifications just as another implementation. The only difference has to be with the execution technique: specifications are executed using term-rewriting, while implementations are executed by an interpreter. This kind of mixed execution may be used to obtain the output for a given input (in order to compare the result against the expected one during testing) or to check if an implementation of a partially specified operation satisfies some property for a given input.

An execution will start as a process to evaluate an expression. Each node of the expression will correspond to a function which may have to be executed interpreting its code or else rewriting it using its specification. It may be necessary to change a call to an operation of an ADT by a call to a routine implementing it or viceversa and some data structures may have to be transformed into the term they represent using the abstraction function. These steps on the evaluation are nothing but changes of subexpressions by equivalent ones.

To illustrate the possibilities of mixed execution (for more details, see [BF95]) we show some examples of evaluation of the expression *succ*(*create2*(1), 1). Let's suppose that an implementation has been selected for *NETWORK*; then, we consider three different scenarios depending on how developed is the implementation

Scenario 1: *create2* is the only implemented function. We assume its codification to be a loop over the matrix' rows, assigning false to each position except for the task identified by the parameter. The evaluation will proceed this way (see figure below): 1) Change the call to *create2* by a call to its implementation, obtaining *abstr*(*create2*$_{IMP\_NET\_I}$(1))[5]. 2) Interpret it to obtain a matrix. 3) Compute the abstraction function, obtaining the term *t*. built up with successive applications of *add*, one for each natural in the range from 2 to MAX. 4) Rewrite *succ(t, 1)*, obtaining a term of type *list_of_natural* which will be interpreted if an implementation for lists has been selected for the result.



Scenario 2: *create, succ* and *add* are implemented; the rest of the functions are not. The specification of *create2* should state that *create2(x)* is equivalent to *add(add(...(add(create, x, 1), x, 2), ..., x, max)*. Now, the evaluation will be: 1) Rewrite *create2* obtaining the term *t* as before. 2) Bottom-up interpretation of *create, add* and *succ*.

Scenario 3: *create2* is the only implemented function. We assume its codification to be first the creation of the empty network and then a loop over *i* from 1 to MAX except for *x*, calling *add(x, i)* in

---

5 f_I stands for the routine of the implementation module *I* implementing *f*.

the body. Steps of the evaluation: 1) interpretation of *create2*. The loop builds up the term *t*. 2) Rewrite *succ(t, 1)*.

5 Conclusions

We have presented in this paper many facets of a single project. In this project (named Excalibur) we are designing a language to specify and implement components and an environment for it, but the language and the environment are not the project goal by themselves, but a pretext to experiment with our methodological ideas. To be precise, we have introduced:

- A formally defined notation to state non-functional issues of software systems. Although some previous work has been done at the process-level (see [MCN92]), we do not know of any approach at the product-level with the same features as ours (in spite of many claims in this sense [Sha84, Win90, MCN92, Jaz95]). There are many non-formalised or partial proposals [Mat84, LG86, Win89, CGN94, SY94] which results are subsumed in our work; also, [CZ90, CZ91] present a very interesting framework close to ours, but restricted to non-functional properties taking numerical values. Our notation leads to a system in which multiple implementations coexists, together with an algorithm to select the best one for a given specification. A few proposals but with important restrictions have been proposed in this direction (the language SETL [Sc+86] or the LIBRA system [Kan86]).

- A software process assistant based on the idea of prototyping with incomplete programs. The guidance trough an assistant for the process model is a common idea in the fields of Software Process Modelling and Process-centered Software Environments ([FKN94]).

- A mixed execution mechanism being able to deal with incomplete systems. This makes our approach well-suited for a prototyping software process. Execution is possible even combining multiple implementations and having incompletely specified operations (which can only be used in restricted queries) or partial implementations. To assure full executability, the abstraction function may be used in some contexts. Some approaches exist in this sense (the execution system in Asspegique [CK90] or the environment for the LEDA multiparadigm language [Bud95]); however, we do not know of any of them as flexible as ours.

Currently, a first prototype of the selection algorithm and the execution system exists. It must be remarked that, instead of having different interpreters for all the programming languages we are interested in[6], we are building a single interpreter of an *ad hoc* notation complete enough to translate implementations in C++, Eiffel, Ada-95, etc., into it; this notation will be used during prototyping only and it will disappear once the final version of the system is obtained. This strategy decreases considerably the task of adapting our system to new programming languages.

References

[BF95] X Burgués, X. Franch. “Evaluation of Expressions in a Multiparadigm Framework”. In *Proceedings of 7th PLILP*, Utrecht (The Netherlands), LNCS 982, Springer-Verlag, 1995.

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

[Bud95] T.A. Budd. “Multiparadigm Programming in Leda”. Addison-Wesley, 1995.

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

[CK90] C. Choppy, S. Kaplan. “Mixing Abstract and Concrete Modules”. In *Proceedings of 12th ICSE*, Nice (France), 1990.

6 The programming languages are O.-O. ones to support the coexistence of multiple implementations.

[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, “Supporting Incremental Component Programming with Functional and Non-Functional Information”. Accepted for publication in *Proceedings of SCCC conference*, Valdivia (Chile), 1996.

[FKN94] A. Finkelstein, J. Kramer, B. Nuseibeh (eds.), *Software Process Modelling and Technology*, John Wiley & Sons ed., 1994

[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, Universitat Politècnica de Catalunya, (Spain), 1996. Available in català (english extended summary currently in progress).

[Hoa72] C.A.R. Hoare. “Proof of Correctness of Data Representations”. In *Programming Methodology*, Springer-Verlag, 1972.

[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.

[Kan86] E. Kant. “On the Efficient Synthesis of Efficient Programs”. In *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[Knu76] D.E. Knuth. “Big Omicron and Big Omega and Big Theta”. SIGACT News, 8(2), 1976.

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

[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 Soft. Engineering, 18(6), 1992.

[RES94] “Special Feature: Component-Based Software Using RESOLVE”. ACM Software Engineering Notes, 19(4), Oct. 1994.

[Sc+86] J. Schwartz et al. *Programming with Sets: Introduction to SETL*. Springer-Verlag, 1986.

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

[SY94] P.C-Y. Sheu, S. Yoo. “A Knowledge-Based Program Transformation System”. In *Proceedings 6th CAiSE*, Utrecht (The Netherlands), 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.