implementation selection. A: International Conference on Reliable Software Technologies. "Reliable Software Technologies, Ada-Europe '97: 1997 Ada-Europe International Conference on Reliable Software Technologies: London, UK, June 2-6, 1997: proceedings". Berlín: Springer, 1997, p. 88-99.
The final authenticated version is available online at https://doi.org/10.1007/3-540-63114-3_9

Including Non-Functional Issues in Anna/Ada Programs for Automatic Implementation Selection

Xavier Franch

Universitat Politècnica de Catalunya
Department Llenguatges i Sistemes Informàtics
Pau Gargallo 5. E-08028 Barcelona (Catalonia, Spain)
e-mail: franch@lsi.upc.es

Abstract. We present an enrichment of the Anna specification language for Ada aimed at dealing not only with functional specification of packages but also with non-functional information about them. By non-functional information we mean information about efficiency, reliability and, in general, any software attribute measuring somehow the quality of software (perhaps in a subjective manner). We divide this information into three kinds: definition of non-functional properties, statement of non-functional behaviour and statement of non-functional requirements; like Anna annotations, all of this information appears in Ada packages and package bodies and their syntax is close to Ada constructs. Non-functional information may be considered not only as valuable comments, but also as an input for an algorithm capable of selecting the "best" package body for every package definition in a program, the "best" meaning the one that fits the set of non-functional requirements of the package in the program.

1 Introduction

Component programming [Jaz95, Sit+94] is a useful and widely employed way of building complex software systems by means of combining, reusing and producing software components. What a component does is stated by its functional properties. Different implementations must satisfy them, but they will differ in some non-functional aspects, such as execution time or reliability.

Among other possibilities, we are interested in software components as an encapsulation of *abstract data types* (*ADT*) [Gut75], described by algebraic specifications and implemented using an imperative or object-oriented programming language. To be more precise, we view software components consisting 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 paper, we propose component programming with ADTs using:

- Ada [Ada83] as the programming language. Then, definitions of ADTs are encapsulated in packages while implementations appear inside package bodies[1].
- The Anna specification language [LH85, Luc90] for stating functional properties of ADTs.
- Some new constructs [Fra96, FB96, FB97] for dealing with non-functionality.

We consider three kinds of non-functional information:

- *Non-functional property* (short, *NF-property*): any attribute of software which serves as a means to describe it and possibly to evaluate it; for instance, time efficiency of a procedure or portability of a package body.
- *Non-functional behaviour* of a component implementation (short, *NF-behaviour*): any assignment to the NF-properties which have been declared of interest for the implemented software component.
- *Non-functional requirement* imposed on a software component (short, *NF-requirement*): any constraint referring to a subset of the NF-properties which have been declared of interest for the software component.

In order to make our approach more attractive, we integrate these last new constructs into the Anna notation, providing then an integrated framework where functional and non-functional aspects of software are uniformly considered, as we think they always should be.

The rest of the paper is structured as follows. We review in section 2 the main features of the Anna specification language. Sections 3, 4 and 5 introduce non-functional properties, behaviour and requirements, respectively. Section 6 shows how our packages are managed to produce different files distinguishing the non-functional part from the functional one. Section 7 gives an outline of the automatic selection algorithm. Finally, section 8 provides the conclusions.

2 The Anna Specification Language

The Anna specification language (ANNotated Ada) [LH85, Luc90] is a language extension of Ada that includes features supporting functional specification such that:

Anna program = Ada program + formal comments

Formal comments are just comments from the Ada point of view, and so Anna programs are acceptable by Ada compilers with no changes at all. However, these comments obey some syntactic rules and they have a semantic meaning. There are two types of formal comments:

[1] Hereafter, we use the words "definition" and "package" interchangeably, and the same with "implementation" and "package body".

- Virtual Ada text. Definition of virtual concepts by means of usual Ada subprogram constructs. These virtual concepts are used in annotations (see below) to state functional specifications, and their definition can be either by a (virtual) Ada body or by means of annotations; in the first case, they are executable as usual Ada subprograms. Syntactically, lines in virtual Ada text begin with '--:'.
- Annotations. Statement of functional specifications, possibly involving virtual concepts introduced with virtual Ada text. There are many kinds of annotations, identified by means of key words. We remark:
  - Object annotations. They constrain the value of one or more program variables.
  - Subtype annotations. They support the formulation of representation invariants [Hoa72] of types.
  - Statement annotations. To formulate arbitrary assertions in the middle of Ada code.
  - Subprogram annotations. A kind of pre post specification of functions and procedures that may include constraints on their formal parameters, function results, etc.
  - Axiomatic annotations. Classical algebraic specification of ADTs by means of conditional equations.

  Syntactically, lines in annotations begin with '--|'.

A package representing mappings (mathematical functions) is shown in fig. 1; we will suppose in the rest of the paper that we have operations to create the mapping, to add a pair <key, information>, to remove a pair given its key, to obtain the information associated to a key and also to get the list of defined keys; to simplify some details that are not relevant to our work, we suppose that both keys and information are integer numbers. We present a private type representation by means of a bounded array with a cursor to the first free position; pairs are stored in order of arrival. So, addition of new pairs must take care of lack of space; for this reason, we introduce two virtual Ada functions, one to compute the number of pairs already in the array and the other to find out if a key is already defined. As part of the type representation, we provide a representation invariant which states the valid values of the cursor (this could be made with Ada code) and also that keys do not appear more than once in the data structure (this part could not be written with Ada code); the last fact uses two quantifiers provided by Anna. Note that, as stated in [Hoa72], some relationships implicitly hold as pre and post conditions of ADT operations, coming from the invariant.

As a convention, we use lowercase identifiers (except for initials) for virtual Ada text, annotations and also non-functional information. For the sake of brevity, we do not include the package body, which could provide code for the virtual Ada subprograms in order to obtain executable annotations (in this case, the code is easy to write and not too inefficient, which could be a problem when executing annotations).

```
package MAPPING_ADT is
    use LIST_OF_INTEGER;  -- for the GET_ALL_KEYS operation
    type MAPPING is private;
    --: function NbKeys (M: MAPPING) return NATURAL;
    --: function DefinedKey (M: MAPPING; K: INTEGER) return BOOLEAN;
    …
        procedure ADD_PAIR (M: in out MAPPING; K, V: in INTEGER);
    --|     where in not DefinedKey(M, K) => (NbKeys(M) <= MAX_KEYS),
    --|               out DefinedKey(M, K);
    …
    private
        type MAPPING is
            record A: array (1..MAX_KEYS) of record k, v: INTEGER end;
                      FREE: INTEGER;
            end record;
    --| where M: MAPPING =>
    --| (M.FREE > 0) and (M.FREE <= MAX_KEYS + 1) and
    --| for all K: INTEGER =>
    --|     (1 <= K < M.FREE =>
    --|             not exist R: INTEGER =>
    --|                     (R <> K) and (M.A[R].k = M.A[K].k))
end MAPPING_ADT;
```

Fig. 1. An Anna specification for mappings

From this description, we may conclude that Anna is a powerful and easy-to-use notation for stating functional specifications of ADTs implemented with Ada. However, it does not provide any mean to deal with non-functional aspects of packages, such as efficiency or reliability. So, in the rest of the paper we enrich Anna with some new constructs to obtain what we think it is a complete specification language, by putting functional and non-functional specifications together.

3 Declaration of Non-Functional Properties

NF-properties attached to software components may be of many different kinds: 1) *boolean*, to represent software attributes which simply hold or fail (ex.: full portability of an implementation); 2) *integer*, to introduce software attributes that can be measured (ex.: reliability of an implementation); 3) *by enumeration*, to represent software attributes which can be classified into some categories (ex.: type of user interface); 4) *string*, to associate arbitrary identifiers as value of NF-properties; 5) *asymptotic*, to establish the execution time and space of types and operations.

Asymptotic NF-properties need not to be explicitly declared; their existence is inferred from the definition: there is an NF-property for every type measuring the space of its representation, and there are two NF-properties for every public procedure, one for its execution time and the other for its auxiliary space. In the ADT framework, we measure efficiency with the *big-Oh asymptotic notation* [Knu76, Bra85], defined as:

$$\mathrm{O}(f) = \{g\colon \mathrm{N}^+ \rightarrow \mathrm{N}^+ \,/\, \exists c_0 \in \mathrm{N}^+,\ \exists n_0 \in \mathrm{N}^+\colon \forall n \geq n_0\colon g(n) \leq c_0 f(n)\}$$ [2]

(N^+ stands for non-zero natural numbers.) Values of this kind of NF-properties are given in terms of some *measurement units*, which represent problem domain sizes and which must also appear in definitions.

A set of possible NF-properties for the *MAPPING_ADT* package is shown in fig. 2. Their names, together with the comments we include, are self-explanatory enough. Some of the numerical and by enumeration properties declare the valid values they can take. In the case of *reliability*, there is an implicit ordering (writing ordering) which allows later to write expressions as "reliability >= medium". We introduce a measurement units for the keys[3]. Note that we declare the properties as virtual Ada text, as is usually done when working with Anna.

```
package MAPPING_ADT is
    ... declaration of interface with Anna annotations
    --: properties
    --: boolean error_recovery, fully_portable;
    --: string supplier;   -- "own" stands for software produced in the company
    --: integer reusability_degree [0..5];
    --: integer month [1..12], year;    -- date of delivery of the component
    --: enumeration reliability = (none, low, medium, high);
    --: measurement units nb_keys;
end MAPPING_ADT;
```

Fig. 2. Declaration of NF-properties for mappings

In order to allow programmers to define their own catalogue of NF-properties, it is possible to introduce them in separate packages which do not contain real Ada code; these packages may be used inside any other one. Although we are not going to show this in the paper, it is possible to form hierarchies with these packages. For instance, we can reformulate the *MAPPING_ADT* package as in fig. 3.

[2] We may extend this definition for the case of having more than one parameter to measure efficiency.

[3] This measurement unit does not have the same meaning as the *NbKeys* virtual function of fig. 1: the first one concerns asymptotic sizes, while the second one counts the actual number of keys.

```
package CREATION_ISSUES is
    --: properties
    --: integer month [1..12], year;    -- date of delivery of the component
    --: string supplier;   -- "own" stands for software produced in the company
end CREATION_ISSUES;

package MAPPING_ADT is
    ... declaration of interface with Anna annotations
    --: properties
    --: use CREATION_ISSUES;
    --: boolean error_recovery, fully_portable;
    --: integer reusability_degree [0..5];
    --: enumeration reliability = (none, low, medium, high);
    --: measurement units nb_keys;
end MAPPING_ADT;
```

Fig. 3. An equivalent declaration of NF-properties for mappings

4 Statement of Non-Functional Behaviour

Each package body implementing an ADT should state its NF-behaviour with respect to the NF-properties declared in the definition package, including efficiency ones. For instance, the behaviour of the implementation for *MAPPING_ADT* using the representation shown in fig. 1 may look as in fig. 4. Note that non-asymptotic NF-properties and also the asymptotic space of the type representation are listed altogether, while efficiency of operations is stated in the operations themselves. See the use of arithmetic operators to state efficiency, which are interpreted in the big-Oh notation [Bra85]; the equality $time(x) = E$ means $time(x) \in O(E)$ (the same for *space*).

```
package body MAPPING_ADT is
    --| behaviour
    --|      error_recovery; fully_portable; reusability_degree = 4; reliability = high;
    --|      month = 8; year = 1996; supplier = "own";
    --|      space(mapping) = nb_keys;
    …
    procedure GET_ALL_KEYS ...
    --| time(GET_ALL_KEYS) = pow(nb_keys, 2); space(GET_ALL_KEYS) = 1;
    …
end MAPPING_ADT;
```

Fig. 4. Non-functional behaviour of an implementation for mappings

5 Statement of Non-Functional Requirements

NF-requirements state constraints imposed on implementations of software components. Syntactically, they are Ada boolean expressions enriched with some *ad hoc* constructs for non-functionality (see examples below). They may appear both in packages and package bodies.

NF-requirements in packages state the conditions that every implementation of the component must fulfil in order to be useful in the system. Put it in other words, they form the non-functional part of the specification of the component. Below, we enrich the *MAPPING_ADT* package with some relationships between NF-properties: execution time of individual operations must not exceed linear cost, while the time of obtaining the list of all the keys is (asymptotically) bounded by the square of the number of keys; also, some constraints about the value of *reliability* are stated. Note that all of these relations are up to the specifier, although sometimes there are relationships which are inherent to the ADT.

```
package MAPPING_ADT is
    …
    --: properties
    --: use CREATION_ISSUES;
    --: ...the same as before
    --: relations
    --|     time(ADD_PAIR, REMOVE_PAIR, LOOK_UP) <= nb_keys;
    --|     time(GET_ALL_KEYS) = pow(nb_keys, 2);
                  -- "pow" means power; here, symbol "<=" stands for set inclusion
    --|     not error_recovery => (reliability <= medium);
    --| (not fully_portable and supplier <> "own") => reliability = none;
end MAPPING_ADT;
```

Fig. 5. Non-functional specification for mappings

NF-requirements appearing in a package body *V* state the conditions that the implementations of all the software components used by *V* must fulfil in order to be useful in *V*. Their purpose is to represent the environment into which implementations are to be introduced. They should be complete enough to select a single package body (i.e., a single implementation) for each imported software component. In the general case, *V* will include a list of NF-requirements over every imported component[4]; the importance of NF-requirements in the list corresponds to their order of appearance.

[4] This corresponds with the usual case of having requirements with different degrees of importance.

For instance, an NF-requirement imposed on lists in the package body for *MAPPING_ADT* could be: first, list implementation must be as reliable as possible; next, the cost of the operations to build a list (*EMPTY* and *PUT*) must be constant (i.e., O(1)); last, list traversal should be as fast as possible. To state this requirement, we can use a few predefined operators, such as *max* and *min*, which have an intuitive meaning.

```
package body MAPPING_ADT is
   use LIST;
   --| behaviour   ...the same as before
   --| requirements on LIST_OF_INTEGER:
   --|        max(reliability);
   --|        time(EMPTY, PUT) = 1;
   --|        min(time(TRAVERSAL));
   ... package body using Ada and Anna features
end MAPPING_ADT;
```

Fig. 6. Non-functional requirements on lists inside *MAPPING_ADT*

6 Package Organisation and Processing

Packages and package bodies are analysed in a pre-process step and they generate two different files:

- An Anna/Ada package, where every appearance of non-functional information has become just a comment. This file can be then processed by a standard Anna toolset.
- A file containing non-functional information in an abstract-syntax tree form. These kinds of files are the ones managed by the selection algorithm presented in the next section.

Some words should be said about the existence of multiple implementations for software components. In standard Anna/Ada programs, it is not possible to have more than one package body for the same package, which contradicts our definition of software component. To solve this problem, we allow more than one pair package - package body to exist for a given ADT. Every pair shares the same identifier in the package (although the name of the files will differ), but they have different identifiers for the package bodies (*MAPPING_BY_LISTS*, *ORDERED_MAPPING*, etc.). To make explicit that a package body is an implementation of a package, we add a new key word, "implements", to appear in the header after the identifier, as in:

package body MAPPING_BY_LISTS **implements** MAPPING_ADT **is**...

So, our tool is able to keep track of the package bodies corresponding to a given ADT, which is necessary to select the best implementation for an ADT in a program in an automatic manner. When generating Anna/Ada packages, headers and identifiers are manipulated to follow Ada conventions.

7 Automatic Selection Algorithm

We have built an algorithm to select implementations of ADTs in a program. If all packages have their non-functional information completely defined, the algorithm may proceed starting from the package body that contains the main procedure down to the hierarchy of packages that form the whole program.

Every time a package body *M* is reached, it is necessary to select which implementations to attach to the packages that *M* uses: for every package *D* used in *M*, the algorithm proceeds by computing the list of NF-requirements for *D* appearing inside *M*. NF-requirements in the list are applied in order of appearance until one of the following conditions holds:

1) a single implementation is selected;
2) applying the next NF-requirement would yield 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, with the result that requirements may have to be reviewed.

The result defines the set of valid implementations for *D*; if there is more than one, NF-requirements for *D* appearing in other packages will choose one of them. The implementation finally selected may also use other packages, which are processed in the same way; so, selection of an implementation becomes selection of a tree of implementations.

The algorithm may fail for any of the following reasons (apart from purely lexical, syntactical or type errors in the source code):

- NF-behaviour has been left incomplete (or even does not exist) in some packages.
- An NF-requirement imposed on a package is not satisfied by any of its implementations[5]. The programmer must decide whether this NF-requirement can be relaxed somehow; otherwise, a new implementation satisfying it must be built.

[5] We consider that a list of NF-requirements is violated if the first NF-requirement in the list is not satisfied by any implementation.

- There is not a single implementation satisfying all the NF-requirements imposed on its definition [Fra94]. The programmer must decide how NF-requirements can be relaxed somehow to obtain an implementation valid in all its contexts of use. Also, the algorithm may be tuned to examine only the first NF-requirement in the lists of NF-requirements, which may enlarge the set of implementations locally satisfying the requirements.

It is worth mentioning that the algorithm may select multiple implementations of the same ADT in different places of the program. This coexistence is allowed, provided that there is no interaction between objects of the same type but different implementations. To make possible this scenario, different copies of the same definition package must be done, with different names to avoid clashes.

8 Conclusions

A language for adding non-functional information of software components into Anna/Ada programs has been presented. Non-functional information is expressed in a consistent way with respect to Anna/Ada programs and complements the functional specification part, which is well covered by Anna. The kind of non-functional information provided is complete enough to support automatic selection of the best implementations of components in every context where they appear.

We think that our work offers two main contributions:

- We provide a complete, easy-to-use (at least, for Ada programmers) and formally defined notation [Fra96] to state non-functional issues of software systems. This notation improves software understandability, reusability and maintenance, since more information appears in the software itself. In spite of many claims to this effect [Sha84, Win90, MCN92, Jaz95], we do not know of any approach providing a programming language with the same features as ours. There are many non-formalised or partial proposals [Mat84, LG86, Win89, CGN94, Sit94, SY94] the results of which are subsumed in our work. Also, [CZ90] present a very interesting framework, close to our selection algorithm but restricted to non-functional properties taking numerical values; they do not integrate their approach into any programming language.
- The best implementations for software components can be automatically selected. The existence of such an algorithm supports software development and also software maintenance [FB97], because non-functional modifications in the environment of the system or in the components themselves (i.e., as implementations become more carefully tested or their efficiency is improved) require no more than re-running the algorithm to update the software. A few proposals have been made in this direction [Sc+86, Kan86], but they are bound to languages with major restrictions (a few implementations for a few types).

Currently, an initial prototype of the selection algorithm exists. This algorithm, in fact, may be applied to any Ada-like programming language, provided that we change the Lex and Yacc files that generate the abstract-syntax tree internal representation from non-functional information.

There is a lot of future work to be done. First, we would like to adapt our approach to Ada95, to support inheritance. Also, we would like to allow interaction between objects of the same type but different implementations. Finally, the constructs concerning non-functionality may be enriched in many ways: by defining derived NF-properties, by building predefined catalogues of NF-properties, etc.

References

[Ada83] U.S. Departament of Defense. *Reference Manual for the Ada Programming Language*. American National Standards Institute/MIL-STD-1815A-1983, 1983.

[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 *Procs. of Interface Definition Languages Workshop*, SIGPLAN Notices 29(8), 1994.

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

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

[FB97] X. Franch, P. Botella. "Supporting Software Maintenance with Non-Functional Information". In *Proceedings of 1st Euromicro Conference on Software Maintencance and Reengineering*, 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, Universitat Politècnica de Catalunya (Catalonia, Spain), 1996.

[Gut75] J.V. Guttag. *The Specification and Application to Programming of Abstract Data Types*. Ph.D. Thesis, University of Toronto, 1975.

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

[Jaz95] M. Jazayeri. "Component Programming - a Fresh Look at Software Components". In *Proceedings of 5th ESEC*, Barcelona (Catalonia, Spain), 1995.

[LH85] D.C. Luckham, F.W. von Henke. "An Overview of Anna, a Specification Language for Ada". Software IEEE, March 1985.

[Luc90] D.C. Luckham. *Programming with Specifications: an Introduction to ANNA, a Language for Specifying Ada Programs*. Texts and Monographs in Computer Science, Springer-Verlag.

[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 Transactions on Software Engineering, 18(6), 1992.

[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.

[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). "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 (The Netherlands), LNCS 811, 1994.

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

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