1987

Normalized Specifications for Identifying Reusable Software

Luqi

IEEE

Normalized Specifications for Identifying Reusable Software

Luqi

Naval Postgraduate School Code 52
Monterey, CA 93943

ABSTRACT

An approach to retrieving reusable software components by means of module specifications is described. The approach depends on normalizing specifications to reduce the variations in the representation of software concepts. The concept is illustrated in terms of both formal and informal approaches to component specifications.

Key Words

Reusable Software, Component Specification, Software Base, Rapid Prototyping

1. Introduction

Reusable software has been identified as a promising means for increasing software productivity [8,9]. Reusing software is especially effective when used together with a rapid prototyping approach to software development [3,4]. An effective way to retrieval reusable software components from a software base [2] is needed for this approach. Two important problems must be addressed to achieve effective component retrieval:

(1) Find all of the components in the software base performing the function requested by the designer;

(2) Find adaptable components with similar functions in cases where the software base does not contain any components corresponding exactly to the retrieval request. This paper is concerned with the first of these problems. An approach to the second problem can be found in [6,7].

The effectiveness of a retrieval scheme can be measured by the difference in effort between finding a reusable component and designing, implementing, and testing a new one for the same function. A proposed method is using module specifications as a basis for retrieval [2]. This method should be effective because the module specifications must be produced anyway in software development projects of appreciable size. The normalization of the specifications for software components must be developed together with the retrieval techniques based on those specifications [6]. None of the previously proposed systems for retrieving reusable software is able to do so based on semantic specifications. Such a facility is critical for the application of reusable software to rapid prototyping, where designer time is restricted.

The essential problem in component specification is to enable efficient retrievals based on specifications without eliminating the expressive power needed for the practical application of black-box specifications in design. The limited designer effort available in rapid prototyping dictates that the same specification must be used both as a design tool and as a basis for computer aided retrievals of reusable components. Different designers think in different ways, and they are likely to reject any notation that allows a given concept to be expressed in only one way, because the rigid thinking style imposed by such a notation would be too cumbersome and unnatural for most of them. However, information retrieval is made much more complex by having many different representations for the same information. Existing methods for information retrieval are based solely on the syntactic form of the descriptions stored with each component, rather than the semantics of the descriptions.

We propose to solve this problem by seeking specifications with a normal form that can be generated mechanically. If many different specifications with the same meaning can be reduced to the same normal form, then designer can have freedom of expression while allowing the information retrieval system to have fewer syntactically distinct forms for each semantically distinct module that may appear in the software base, since they can be unboundedly many syntactic forms for the same semantic description, reduction to normal form is a more practical approach than attempting to generate all variations and searching the software base for each variation. Our approach requires normalized component specifications to be stored in the software base along with the implementations of the reusable components. Component specification in queries must also be normalized before being submitted to the software base management system. Two kinds of normalization techniques for specification are discussed respectively in section 2 and 3.

2. NormalizingInformal Specifications

Informal specifications are easy for people to use, but they are difficult for machines to process. The normalization transformations that can be applied to natural language specifications are either shallow or require automated understanding of natural language. The shallow approaches are not strong enough in the sense that there are many equivalent descriptions that cannot be reduced to the same normal form by means of syntactic transformations. Programs for understanding natural languages are very difficult to build. Standardizing terminology is one way to normalize informal specifications. This can be done by using a synonym table and a text substitution tool (e.g. the *sed* stream editor of Unix). An

CH2468-7/87/0000/0046$01.00 © 1987 IEEE

example of a fragment of a synonym table is shown below.

TERM	ALIASES
update	change, modify, refresh, replace, substitute
read	fetch, obtain, input, get, retrieve

The transformation defined by such a table simply replaces all occurrences of the aliases by the associated basic terms given in the table. For example, the sentence "Fetch the order from the transaction file and modify the inventory" would be transformed to "Read the order from the transaction file and update the inventory" This kind of approach has the virtue of being easy to implement. It has the disadvantages of introducing subtle changes of meaning and of still leaving many syntactically different ways of expressing the same idea, lowering the probability that a component in the software base will be found based on an independently constructed description of its function. This kind of transformation changes names, but preserves the structure of the original statements, so that individual stylistic differences will result in distinct normalized specifications, even though they may be paraphrased versions of the same statement. Nevertheless, this simple approach may have some practical usefulness in the early stages of requirements analysis where the dominant representation is English text.

Another approach uses a natural language parser to produce a frame-based representation of the objects and relationships described by the informal specification. A potential advantage of such an approach is to allow different styles and sentence structures to be normalized to the same representation. The disadvantages of this method are that it is expensive, requires specialized skills to implement, and is difficult to apply unless the subject matter is restricted to a domain with a small vocabulary. Furthermore, the ambiguities inherent in natural language remain, resulting in the retrieval of components that are not relevant to original specification.

A more practical approach is to give up trying to model the precise meaning on the informal specification, and to rely on keywords to try to capture an approximate set of relevant components. A problem with this approach is assigning keywords to modules. Manual approaches to classification such as [7] are error prone and may require a relatively large investment for assembling a large software base. This has been avoided in [1] by using a vector of term frequencies in the document instead of manually chosen keywords. However, the resulting uncontrolled vocabulary leads to more false retrievals and requires an interactive session to adjust weighting factors until a suitable ranking of candidate components can be obtained. The effort required in both approaches for weeding out false retrievals makes informal specifications unattractive as a basis for component retrieval supporting rapid prototyping.

3. Normalizing Formal Specifications

Formalized specifications are subject to stronger transformations, which can reduce two specifications to the same normal form even in cases where they have different structures, reflecting different conceptual approaches to describing the problem. We illustrate these transformations by means of an example. A specific syntax is needed in order to show the example. We use ordinary mathematical notations here, to make the examples easy to follow, and we do not intend to imply that the same representation will be used by the programs for normalizing specifications. Consider the two specification fragments shown below, both of which record the requirement that the sequence *REPLY* must be sorted in increasing order.

A: 1 <= i < j <= length(REPLY)
    => REPLY[i] <= REPLY[j]

B: REPLY = a @ [x] @ b @ [y] @ c => x <= y

Specification A uses indices in the REPLY sequence to describe the required ordering, while specification B describes the same ordering in terms of subsequences and the concatenation operator "@". Logical implication is denoted by "=>" and the sequence of length one containing the element x is denoted by "[x]". The REPLY keyword is a constant with a special interpretation, representing the output value of a software module.

The transformations and simplifications that can be performed on such specifications depend on knowledge about the the properties of the operations on the underlying data types. These properties can be expressed as conditional rewrite rules to make the simplification process easier. For example, the relationship between indices and the data value at a given position in a sequence is described by the following rule.

R1: s = a @ [x] @ b => s[length(a) + 1] --> x

This rule says that the index of x in the sequence s is length(a) + 1, which follows from the convention that the index of the first element of a sequence is one. The notation "a --> b" means a = b, with the additional directive to substitute b for a in the simplification process, but not vice versa.

Rule R1 can be applied to specification A under the substitutions (s: REPLY, i: length(a) + 1) to give the reduced specification

A1: REPLY = a @ [x] @ b
    & 1 <= length(a) + 1 < j <= length(REPLY)
      => x <= REPLY[j]

Rule R1 can be applied again, to A1 with the substitutions (s: REPLY, j: length(c) + 1) to give

A2: REPLY = a @ [x] @ b
    & REPLY = c @ [y] @ d
    & 1 <= length(a) + 1 < length(c) + 1
      <= length(REPLY)
        => x <= y

At this point, some more rules describing the properties of the "<" operator are needed.

R2: x < y + x --> 0 < y

R3: x <= y + x --> 0 < y

R4: 0 <= length(s) --> true

R5: true & p --> p

R6: p & true --> p

R7: x <= y < z --> x <= y & y < z

R8: x < y <= z --> x < y & y <= z

Rules R2 and R3 are facts about the standard ordering on integers, while rule R4 is a theorem about lengths of sequences, expressed as rewrite rules. Rules R5 and R6 are standard absorption laws of boolean algebra. Rules R7 and R8 define repeated inequalities by the usual conventions. The condition

1 <= length(a) + 1

is reduced to true by rules R2 and R4, and eliminated from A2 using R7 and R5. The rules

R10: REPLY --> c @ [y] @ d.

R11: length(s @ t) --> length(s) + length(t)

R12: length([x]) --> 1

R13: x + y <= z + y --> x <= z

are relevant at this point. R10 is derived from one of the other equations in the hypothesis of the implication. R11 and R12 are basic facts about lengths of sequences, and R13 is another standard inequality law. The condition

length(a) + 1 < length(c) + 1

is simplified to

length(a) < length(c)

by R13. The condition

length(c) + 1 <= length(REPLY)

can be reduced to true by applying rules R10, R11 (twice), R12, and then R3 and R4. The condition is eliminated from the implication entirely by R6. The result of these simplifications is the following.

```
A3: REPLY = a @ [x] @ b
    & REPLY = c @ [y] @ d
    & length(a) < length(c)
      => x <= y
```

Further progress can be made by R14, the common prefix law for sequences.

```
R14: length(s) < length(u) & s @ t = u @ v
       => u --> s @ w
```

Under the substitutions (s: a @ [x], t: b, u: c, v: [y] @ d) this leads to

A4: REPLY = a @ [x] @ w @ [y] @ d => x <= y

which is the same as specification B, up to renaming of variables. Variable names can easily be standardized, by picking them from a fixed list in order of occurrence in the formula. The result of doing that to either A4 or B is shown below.

```
A5: REPLY = x1 @ [x2] @ x3 @ [x4] @ x5
      => x2 <= x4
```

A5 may be less readable to a human than A4, but is quite suitable as a basis for automated retrieval.

4. Conclusions

Formal specifications appear to be more suitable as a basis for the retrieval of reusable software components than informal specifications. Formal specifications are free from the ambiguity inherent in natural language specifications because formal languages used have been expressly designed to avoid ambiguity. Using predicate calculus as the formal language has the advantage of bringing to bear a well studied area of mathematics, namely logic and the theory of term rewriting systems. These systems bring with them more powerful transformations that preserve the meaning of a sentence while dramatically affecting its form. Since many formal specification languages are close to predicate calculus, it is relatively straight forward to map such a specification into first order logic. The specification for the reusable components in a software base can either be written directly in predicate calculus, or they can be written in some other formal specification language and mechanically translated into predicate calculus. The latter approach has the advantage of enabling the same software base management system to accept components with specifications in a variety of formal languages, allowing more effective use of existing module specification. In such an approach, each module would have an implementation and two different specifications, one for human consumption, and a mechanically derived normalized form that would be used only by the component retrieval system.

More work is needed to develop simplification rule systems that are strong enough to standardize many common ways of expressing the same concepts, while still remaining disciplined enough to allow a uniform guarantee of termination. Such simplification systems are needed for all of the data types commonly used in specifications. A uniform approach to constructing such systems is needed to properly handle user defined data types, since the set of types used in practice is extensible. Since the general word problem in algebra is undecidable, it is not reasonable to expect a perfect solution to the problem, which would be a system that reduces two specifications to the same normal form whenever they have the same meaning. However, a normalization technique does not have to be perfect to be useful for component retrieval. It suffices to be able to reduce commonly occurring variations of a specification to the same normal form most of the time. Furthermore, many of the data types in common use do have simplification systems that lead to unique normal forms. It is reasonable to expect to be able to find normalization systems that are strong enough to be useful for specification based retrieval of reusable software components. This approach is especially useful as a practical aid to rapid prototyping [5].

Another subject that deserves further attention is the development of heuristics that allow some transformations that expand a term rather than simplify it under some circumstances, but still guarantee termination of the simplification process. An example of such a situation is the application of R10 in going from A2 to A3 in the previous section. Such steps appear to be necessary to enable reductions of substantially different approaches to specifying a concept to the same normal form.

1. C. Landauer and C. Mah, "Message Extraction Through Estimated Relevance", in *Proc. of the Second International Conf. on Information Storage and Retrieval*, ACM, Dallas, 1979, 64-70.

2. Luqi, "Rapid Prototyping for Large Software System Design", Ph.D. Thesis, University of Minnesota, 1986.

3. Luqi and V. Berzins, "Rapid Prototyping of Real-Time Systems", *Revised for IEEE SOFTWARE*, 1987.

4. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *to appear in IEEE TSE*, 1987.

5. Luqi, "Research Aspects of Rapid Prototyping", NPS 52-87-006, Computer Science Department, Naval Postgraduate School, 1987.

6. Luqi and V. Berzins, *A Knowledge Base for Retrieval Reusable Software*, To be submitted to ACM-IEEE 1987 Fall Joint Computer Conference, Dallas, Texas, October 1987. not written yet.

7. R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability", *IEEE Software 4*, 1 (Jan. 1987), 6-16.

8. R. T. Yeh, R. Mittermeir, N. Roussopoulos and J. Reed, "A Programming Environment Framework Based on Reusability", *Proc. Int. Conf. on Data Engineering*, Apr. 1984, 277-280.

9. R. T. Yeh, N. Roussopoulos and B. Chu, "Management of Reusable Software", *Proc. COMPCON*, Sep. 1984, 311-320.