

SOFTWARE COMPONENT REUSABILITY

PRIYANKA DEV & ROHIT SACHDEVA

DEPARTMENT OF COMPUTER SCIENCE

M.M MODI COLLEGE, PATIALA

PRIYANKA.IN@GMAIL.COM, SACHU_147@YAHOO.COM

Abstract- Reusing the existing systems can decrease the time spent building a large system and thus decreases cost since it can significantly reduce the new effort that must be applied for building the component from the scratch. Further reusing components from a well-tested library can reduce the time spent debugging and improve reliability. The key in software reuse is the component selection. It should be faster and easier to locate or retrieve a component from the software repository. There are several techniques with the aid of which we can select the component for reuse. One of the techniques that is gaining popularity these days is formal specification matching. Specification matching is the way to compare two software components based on the specifications of the component's behavior. In the context of software reuse and library retrieval, it can help determine whether one component can be substituted for another or how can one be modified to fit the requirement of the other. The match can either be exact or approximate. These matches capture the notions of generalization, specialization and reusability of software components. In the present work, a technique for retrieving the components from the repository has been implemented using the formal specifications, with an emphasis on relaxed various kinds of matches have been defined with varying degrees of relaxation. Because these formal specifications are pre- and post- written as predicate in first order logic, therefore, they rely on theorem proving to determine match and mismatch.

What is Software Reuse?

Software Engineering as a discipline of computer science emerged after two decades after the invention of digital computers as a reaction to some peculiar failures in the development of large software systems. One of the principles of software engineering is to build new artifacts prototypes, products or systems on

scratch", the importance of this principle had been realized from the very beginning of the discipline. But what actually is software reuse? Various definitions are given for software reuse. These are:

"Software reuse means reusing the inputs, the processes and the outputs of the previous development efforts"

"Reuse is the use of previously acquired concepts and objects in the new situation, it involves encoding development information at different levels of abstraction, storing this representation for future reference, matching of new and old situations, duplication of already developed objects, and their adaptation to suit new requirements."

Software reuse is a mean toward an end, improving software productivity and software product quality. Reuse is based on the premise that educing a solution from the statement of the problem involves more effort than inducing a solution from that to a particular similar problem for which the efforts have already been expended. As software systems are becoming larger and more complex, the demands for high levels of reliability, quality and productivity is also increasing. For the last three decades, the productivity and quality is increasing, but it is not enough to close the gap between the demands placed by industry and what the state of practice can deliver. Software reuse offers a great deal of potential in terms of both quality and productivity by :

- Dealing with the software products at component level and by focusing on arbitrarily abstract descriptions of the software components.

- Dealing with the software design at the architecture level rather than at the coding level.

Reuse Library / Knowledge base for software reuse

A controlled collection of reuse artifacts constitutes a reuse library. Such libraries must not only contain reusable components but are also expected to provide certain types of services to their users e.g. storage, searching, inspecting and retrieval of artifacts from different application domains and of varying granularity and abstraction, loading, linking and invoking of stored artifacts, specifying abstract relationships . The major problems in the utilization of such reuse libraries are in determining appropriate artifact classification schemes and in the selection of methods to effectively and efficiently search the library. But before constructing the software with reusable components, they should have a share of characteristics, which inherently and actively promote the reusability.

Component Retrieval Problem

Given a set of requirements, the first step is of finding a component that satisfies the requirements, either in its present form, or modulo minor modifications. When the number of components in the library is large, developers can no longer afford to examine and inspect each component individually to check its appropriateness. An automated method is needed to perform at least a first-cut search that retrieves an initial set of potentially useful components. Such a method would match an encoded description of the requirements against encoded descriptions of the components in the library. The choice of the encoding methods, for both the requirements and the components and of the matching algorithms involves a number of trade offs between complexity and retrieval quality.

Classifying and Retrieving Components

Consider a large repository of reusable software components. Assuming the availability of these components, they must still be organized so that they can be found by a software engineer when they are required. This is the role of the domain engineer. A key issue in navigating reuse libraries is therefore: How do software components can be defined in an unambiguous, classifiable terms?

Describing Reusable Components

An ideal description of a software component encompasses a 3C Model concept, content and context.

- **Concept:** A description of what the software does. The interface and the semantics of the component are fully described. The concept should communicate the intent of the component.

- **Content:** A description of how the component is realized. Generally, the content information is hidden from casual users and need be known only to those who intend to modify the component.

- **Context:** The placement of a component within its domain of applicability. By specifying conceptual, operational, and implementation features, the context enables a software engineer to find the appropriate component to meet application requirements.

Formal Specifications

Formal Specification matching is a way to compare two components based on the descriptions of the component behavior. In the context of software reuse and library retrieval, it can help to determine whether one component can be substituted bar another or how it can be modified to fit the requirements of the other. Hence, the key process is the component selection. For example, a simple, but widely

used, criterion is the text hatching (keyword). However, keywords cannot convey significantly useful information, they are widely accepted terminology. In practice, library science methods appear most popular with organizations that practice software reuse. Yet, as the size of software libraries increases, and as components grow increasingly complex, it becomes increasingly difficult to trust the retrieval tasks to these informal procedures. Further weakening the case of library science methods is the fact that their success depends critically on human expertise and human experience. A more informative criterion may be based on signatures (syntax and type information). Although signatures encapsulate type information, they still fail to capture in the behavior of component precisely. Natural language descriptions may document semantic information of components. However the inherent ambiguity of natural languages may make it difficult to locate the right component. Here, then the role of formal specification comes into picture. Software components are represented in the software library by formal specifications that describe their functional properties. Because such specifications may be arbitrarily abstract, they allow us to focus the descriptions on those of the components that are most relevant in the retrieval operation. This the chances of a match, as it recognizes the relationship between the two components that have same important properties, but differ in minor details.

There are different kinds of semantic matches given. The paper explores not just exact match between components, but many notions of relaxed match to be specific and to narrow the focus of what match could mean the following assumptions are made:

- The software components in which discussed are functions e.g. C, routines, Ada Procedures, ML functions. These components might typically be stored in a program library, shared directory of files, or software repository.
- Associated with each component. C is a signature Csig and a specification of its Behavior, Cspec.

Where as signatures describe a component's type information which is usually statically checkable, specifications describe the

components dynamic behavior. Specifications more precisely characterize the semantics of a component than just its signature. In this thesis, our specifications are formal, i.e., written in a formally defined assertion language.

Given two components,

$C = \langle Csig, Cspec \rangle$ and $C' \rightarrow C'sig, C'spec$
 The definitions of a generic component match predicate is given as, Match

Definition Component Match

Match: component, component \rightarrow Bool
 $Match(C, C') = Match\ sig(Csig, C'sig) \wedge Match\ spec(Cspec, C'spec)$

Function Specification Matching

For a function specification, S, the pre- and post-conditions is denoted as Spre and Spost, respectively. Spre defines the interpretation of the function's specification as an implication between the two: Spre = Spre \Rightarrow Spost. This interpretation means that if Spre holds when the function specified by S is called, Spost will hold after the function has Executed (assuming the function terminates). If Spre does not hold, there are no guarantees about the behavior of the function. This interpretation of a pre- and post-condition specification is the most common and natural for functions in a standard programming For example, for the Stack top function.

The pre-condition top p is not (isEmpty (s)).

. The post-condition top post is e = last (s).

. The specification predicate top pred is (not (isEmpty (s))) \Rightarrow (e last (s)).

To be consistent in terminology with our signature matching work, the function specification matching is given in the context of a retrieval application. Example matches are between a library specification S and a query specification Q. It has been assumed that variables in S and Q have been renamed consistently. For example, if we compare the stack pop function with the Queue rest function, we must rename q to s

and q_2 to s_2 . The examples presented in this section are intended primarily as illustrations of the various match definitions.

In this section several definitions of the specification match predicate (match spec (S, Q)) has been examined. The definitions are characterized as either grouping pre-conditions S_{pre} and Q_{pre} together and post-conditions S_{post} and Q_{pre} together or relating predicates S_{pred} and Q_{pred} . Both of these kinds of matches have a general form.

Pre/Post Matches

Pre/post matches on specifications S and Q relate S pre to Q pre and S post to Q post. Each match is an instantiation of the generic pre/post match. We consider five kinds of pre/post matches, beginning with the strongest match and weakening the match by relaxing the relations R_1 and R_2 from \Leftrightarrow to \Rightarrow , by adding S_{pre} to S , or by dropping the pre-condition term. In each case, relaxing the match allows us to make comparisons between less closely related components, but weakens the guarantees about the relationship between the two components. For example, dropping the pre-condition term would allow us to relate components that have the same behavior for the subset of inputs that they handle but that make different assumptions about which inputs are valid e.g., routines on arrays with different bounds. However, since we are not comparing the pre-conditions at all, we cannot guarantee that the components are behaviorally equivalent for all inputs.

Exact pre/post Match

If exact pre/post match holds for two specifications, the components are essentially equivalent and thus completely interchangeable. Anywhere that one component is used, it could be replaced by the other with no change in observable behavior. Exact pre/post match instantiates both R_1 and R_2 to \Leftrightarrow and S to S post in the

generic pre/post match. Two function specifications satisfy the exact pre/post match if their pre-conditions are equivalent and their post-conditions are equivalent.

Guarded plug-in match

In some cases, the post condition relation, $S_{post} \rightarrow Q_{post}$ only holds for values of the input allowed by the pre-condition. For example, the last clause mentioned in the post-condition of stack pop is not defined for the empty stack. The guarded plug-in match adds S_{pre} as an assumption (or "guard") to the post-condition relation, to exclude such cases. We instantiate R_1 and R_2 to \rightarrow in the generic pre/post match, as with plug-in match, but we use $S = S_{pre} \wedge S_{post}$ rather than $S = S_{post}$. We use S_{pre} and not Q_{pre} since S_{pre} is likely to be necessary to the conditions under which we try to prove $S_{post} \Rightarrow Q_{post}$.

Guarded Post Match

As with plug-in match, we define a more relaxed guarded match by dropping the pre-condition relation term. Because we do not have the pre condition term, there is no guarantee that S_{pre} actually holds, so we may have to provide an additional "wrapper" in our code to establish S_{pre} before we call the function specified by S .

Exact Predicate Match

We begin with exact predicate match. Two function specifications match exactly if their predicates are logically equivalent (i.e., R is instantiated to \Leftrightarrow). This is less strict than exact pre/post match, since there can be some interaction between the pre and post conditions (i.e., match E-pre/post \Rightarrow match E-pred). In fact, in cases where $S_{pre} = Q_{pre} = \text{true}$, exact pre/post and exact predicate matches are equivalent.

Generalized Match

Generalized match is an intuitive match in the context of queries and libraries: specifications of library functions will be

detailed, describing the behavior of the functions completely, but queries can be simple. The query can focus on just the aspect of the behavior that we are most interested in or that we think is most likely to differentiate among functions in the library. Generalized match allows the library specification to be stronger (more general) than the query and hence allows for simple queries, R in the generic predicate match is instantiated to \Rightarrow . Generalized match is a weaker match than plug-in match (i.e., $\text{match plug-in} \Rightarrow \text{match gen-pred}$).

Specialized Match

Specialized match is the converse of generalized match. $\text{Match spcl-pred} (S, Q) = \text{Match gen-pred} (Q, S)$. A function whose specification is weaker than the query might still be of interest as a base from which to implement the desired function. Specialized match allows the library specification to be weaker than the query, we instantiate R in the generic predicate match to \Leftarrow .

Conclusions

In the presented work, two kinds of matches have been discussed, exact pre / post match. These matches are implemented using the formal specification technique with an emphasis on relaxed match. The implementation shows that the formal specifications are better than the other techniques like library science, hypertext etc.

Though the notion of specification match was originally motivated by the software library retrieval application, it can be also more applicable to other areas of software engineering, for example, determining subtyping in designing class hierarchies, or showing that one component may be substituted for another when upgrading the system. Additional advances in the field of theorem proving would improve this further. For example, in the retrieval domain, we use a less-expensive match, such as signature matching, to initially prune the search space, and then select one

or two potential matches that we verify using specification matching. Specification matching also has the potential to be extended to apply to other problems such as interoperability.

We can invert the notion of specification match: Determining that two components do not match is determining that they mismatch. Hence, a promising direction of future work is to extend our formal framework from the function level to module level and the architectural level by modeling the various kinds of architectural mismatch.