Approximate Retrieval of Incomplete and Formal Specifications applied to Vertical Reuse *

Rebeca P. Díaz Redondo, José J. Pazos Arias, Ana Fernández Vilas and Belén Barragáns Martínez
Departamento de Enxeñería Telemática. University of Vigo. 36200 Vigo. Spain
{rebeca, jose, avilas, belen}@det.uvigo.es

Abstract

This paper describes how ARIFS tool (Approximate Retrieval of Incomplete and Formal Specifications) provides a suitable reusing environment to classify, retrieve and adapt formal and incomplete requirements specifications. Both classification and retrieval tasks are based on functional similarities according to structural and semantic closeness. To this effect, we define four partial orderings among reusable components and different measures to quantify functional differences among them. By using these measures we are able to offer an approximate and efficient retrieval, without formal proofs, and to predict adaptation efforts to satisfy the required functional specification. This paper focus on semantic similarities as they are more remarkable in vertical reuse, that is, the reuse of software within same domain or application area.

Keywords: software reuse, component-based requirements engineering, reuse of formal requirement specifications, iterative and incremental software processes.

1. Introduction

Reusing at early stages of the development process —like at the requirements specification stage— is widely accepted as a desirable aim, because of the possibility of increasing the reuse benefits [10]. However, there is little evidence in the literature to suggest that software reuse at requirements specification stage is widely practiced. Our proposal [5, 4] deals with this concern, offering a methodology to reuse high abstract level components: incomplete specifications —obtained from transient phase of an iterative and incremental requirements specification process—; and their verification results —obtained from a model checking algorithm. In this paper we focus on describing the formal basis and the methodology set up by ARIFS tool, and in [6] it is detailed our proposal to reuse formal verification information in the same context.

The paper is organized as follows: next section summarizes existing works about managing reusable components; section 3 outlines the software development process where software reuse is going to be included; the methodology and objectives of ARIFS tool are explained in section 4; section 5 describes several functional relationships among reusable components and, as consequence, different lattices used to classify them in the repository; the retrieval process is explained in section 6 and reflected in the example of section 7; in section 8 the selection and adaptation of reusable components is explained; and, finally, a brief summary and future work are exposed in section 9.

2. Related work

Organizing large collections of reusable components is one of the main lacks in software reuse, because providing efficient and effective mechanisms to classify and retrieve software elements from a repository is not an easy problem. Retrieving mechanisms usually rely on the same idea: establishing a profile or a set of component’s characterizing attributes which is used to classify and retrieve them from a repository. Whenever this profile is based on formal specifications, problems derived from natural language are avoided, and this formal description is used for establishing a specification matching. The typical process starts expressing the relation between two components by using a logical formula. Then, a theorem prover checks its validity, and only if the prover succeeds, the component is considered to be suitable. The vast number of proof tasks makes a practical implementation very hard, so in many of the following works this number is usually reduced applying other techniques.

In [15] the retrieval process of code components is based on the Larch/ML specification language for component description and the associated interactive Larch prover for retrieval. Formal proofs are restricted to a small subset of the
repository, which is previously selected by using preconditions and postconditions.

REBOUND (REuse Based On UNDERstanding) tool [13] is also based in Larch language, although specification matching is based on the HOL prover, which is almost automated. In order to reduce formal proofs, different heuristics, based on the semantic of specifications, are applied. In this first step, a dynamic ordering is needed, which reduces the efficiency of the retrieval scheme.

In [12] a two-tiered hierarchy of the repository based on formal specifications using OSPL is proposed. The lower level is based on generality relationships; and the higher one on similarity relationships, which are assessed by a clustering algorithm. Firstly, it is selected the most suitable cluster, and secondly, a theorem prover is used to finish the search; besides this, LOTOS [2] is used as a supplement to the functional descriptions in order to take into account the architectural properties of components.

NORA/NAMMR tool [14] is basically a filter pipeline trying to ensure a plug-in compatibility. There are used signature matching filters, rejection filters (based on model checking techniques), and, finally, confirmation filters (based on Setheo theorem prover). One of its main problems is the recursive specifications management which are not supported by Setheo.

Applying theorem proving in the retrieval process is very difficult to automate because certain small parts of proofs (“trivial” or “analogous to”) that human skip, must be explicitly set up in the tool. So one possible solution is increasing the interaction with users (an user-guided solution) to reduce the computational charge. In [7] Fisher proposes library browsing —using a special navigator— as an alternative to library retrieval. On the other hand, having a more automated tool, it is almost impossible to cover all the aspects of the process in a reasonable period of time.

3. Context

In this section we briefly describe the software development process, SCTL-MUS methodology [12], where the reusing environment is going to be included. This methodology joins: on the one hand, the totally formalization of the process, combining different FDTs (model-oriented and property-oriented); and, on the other hand, an incremental and iterative point of view.

In figure 1, the first phase of this methodology (Initial goals) is shown, where a complete and consistent functional specification of the system is obtained from user’s specification. In every iteration of this stage, the user identifies and specifies a set of functional requirements which lead to a growth in the system functionality. These requirements are verified in the current model or prototype to check: if the model already satisfies the requirements; if it is not able to provide these functional requirements nor in the current iteration neither in future ones (inconsistency); or, if the system does not satisfy the requirements, but it is able to do it (incompleteness).

Formal description of functional requirements is made by using the many-valued logic SCTL [12] (Simple Causal Temporal Logic) —in box labeled as SCTL in figure 1. A generic causal requirement in SCTL follows this pattern:

Premise ⇒ ⊗ Consequence,

which establishes a causing condition (premise); a temporal operator determining the applicability of the cause (⇒ ⊗); and a condition which is the effect (consequence). Apart from causation, SCTL is a six-valued logic, even though it is only possible specifying three different values: possible or true (1), non possible or false (0) and unspecified (½).

This concept of unspecification is specially useful to deal with both incomplete and inconsistent information obtained by requirements capture, because although events will be true or false at the final stage, in intermediate phases of the specification process it is possible that users do not have enough information about them yet, so these events are unspecified in these phases. Syntax of an SCTL requirement \( R \in R_{SCTL} \) is as follows:

\[
\begin{align*}
\langle R_{SCTL} \rangle :&= R \land R | R \lor R | R \Rightarrow \odot R | \neg R | a \in \Lambda \\
\Rightarrow \odot :&= \Rightarrow | \Rightarrow \odot | \Rightarrow \odot \odot
\end{align*}
\]

where \( \Rightarrow \odot \) is the set of temporal operators; \( \theta \in \Theta ::= \{true | false \} \) are propositional constants; and \( a \in \Lambda \) is the alphabet of events in a prototype. Temporal operators \( \{\Rightarrow, \Rightarrow \odot, \Rightarrow \odot \odot\} \) —referred to as simultaneously, previously and next—are used to reason about transition successors and predecessors of a given state by determining the order pattern between the state in which premise is formulated, and the states in the scope of the consequence.

![Figure 1. SCTL-MUS methodology.](image)

SCTL requirements are synthesized to obtain a model of the system by using MUS (Model of Unspecified States).
This state-transition formalism allows prototyping and feedback with users—in box labeled as MUS in figure 1—and supports the consistency checking by using a model checking algorithm—in box labeled as Verification SCTL-MUS in figure 1. MUS graphs are based on typical labeled-transitions graph, but including another facility: unspecification of its elements. In figure below an example of MUS graph is shown.

\[
E_0 \xrightarrow{a} a_{\text{uns}} \xrightarrow{e} E_1 \xrightarrow{c} E_2 \xrightarrow{\neg d} E_3 \xrightarrow{g_1}
\]

This system evolves from one state into another when an event or an observable action from \( \Lambda = \{a, b, c, d, e\} \) occurs. In the initial state, \( E_0 \), event \( a \) is specified as a possible one, that is, system \( g_1 \) evolves from this state into state \( E_1 \) whenever event \( a \) occurs. System \( g_1 \) evolves from \( E_0 \) into state \( E_2 \) through an event which has not been specified yet, which is denoted by \( a_{\text{uns}} \). In subsequent iterations the user may specify this transition with a possible event from \( \Lambda \) with the exception of event \( a \), because MUS graphs are deterministic ones. In state \( E_2 \), event \( d \) is a non possible one, which is denoted by \( \neg d \), and, finally, state \( E_3 \) is a totally unspecified state because every event in \( \Lambda \) has not been specified in this state nor as a possible event neither a non possible one\(^1\).

The second stage of the development process, or Refinement phase, starts from the complete and consistent specification of the system requirements obtained in the previous stage, or Initial goals, and its main goal is to design the system architecture. This architecture is expressed by a constructive FDT (LOTOS [8]) which allows a description of the system components, their interactions and interfaces. Finally, Maintenance stage is turned into a development phase whose starting point is the MUS model of the current system.

4. ARIFS: methodology and objectives

Generally speaking, ARIFS tool provides a friendly environment to classify, retrieve and adapt reusable components in the requirements specification phase of the SCTL-MUS methodology. These reusable components gather both its functional specification, which is expressed by the set of SCTL requirements and modeled by the temporal evolution MUS graph, and an interface or profile information, which is automatically obtained from its functional characteristics to classify and retrieve it from the repository (section 5). Besides this, every reusable component stores verification information, that is, the set of properties which had been verified on the MUS graph and their verification results [6].

The main goal is reusing already developed MUS prototypes which are functionally close to the functionality required by the user in order to reduce, on the one hand, synthesis tasks to obtain the current prototype and, on the other hand, future verification tasks because of the verification results linked to each reusable component.

As the software process where the reuse environment is going to be included is totally formalized, we have opted by a formal specification of reusable components. Although in many of previous approaches these formal representations are only used as a pattern to recover low abstract level components (like code), in our approach, these formal specifications are just the content of the components, so we have a content-oriented retrieval. That is, each component is, simultaneously, index and content of the retrieval, and we can reuse high abstract level components.

As our proposal starts precisely from the necessity of minimizing verification tasks in the software development process, we do not apply formal verification in the retrieval process, which entails an exact retrieval. Instead of this, we propose an approximate components retrieval which allows selecting, in an efficient way, suitable components which are functionally close to the query. This approximate retrieval is based on the concept of unspecification, inherent to incomplete systems—which are obtained from a transient phase of the iterative and incremental development process—that is, not everything is true or false, maybe non specified yet. So, although the recovered components do not match all the requirements of the query, it is possible matching them after making some changes. The convenience of reusing the recovered components is decided after analyzing the adaptation tasks needed in each case.

We have also defined a two-step retrieval process: firstly a rough search phase, where a little set of suitable components is retrieved; and secondly, a more refined one, where these components are ordered depending on the adaptation efforts of each one to satisfy the functionality required by the query. The main reason of supporting a layered retrieval process is basically an efficiency one. Dividing the retrieval process in two phases, we merge the two most important tendencies in managing reusable components: static and dynamic ordering. The stiffness of the classification lattice and the big amount of information needed to reuse components in a efficient way are the main drawbacks of static management. Using dynamic management, on the other hand, implies the whole reorganization of the repository according to the functionality specified by the query, whenever a new query is proposed. In spite of its high precision, the retrieval tasks are much slower in this case than

\[^1\]Unspecified events of a state are not represented, only \( a_{\text{uns}} \) of \( E_0 \) because it implies an evolution of the model.
in a static ordering. Having the two-tiered retrieval process, we are able to avoid the main problem of dynamic ordering—the whole ordering of all components in the repository in each query—without losing the high accurate that this mechanism offers.

5. Static classification of reusable components

Establishing functional relationships among components enables defining component hierarchies or lattices to classify and retrieve them in an proper way. In this section, the four functional relationships among components that we have identified are defined. Each one is characterized in terms of a function \( O \) that associates with every MUS graph \( g \) a set \( O(g) \) which constitutes the observable behaviour of \( g \). For every such \( O \), the equivalence relation \( =O \subset G \times G \) is given by \( g =O g' \iff O(g) = O(g') \), and the preorder \( \subset O \subset G \times G \) by \( g \subset O g' \iff O(g) \subset O(g') \), that is, \( \subset O \) provides a partial order between equivalence classes or graph sets indistinguishable using \( O \)-observations, so \( (G, \subset O) \) is a partially ordered set, or poset. A subset \( G_1 \subset G \) is called a chain if every two graphs in \( G_1 \) are \( O \)-related\(^2\). Two non \( O \)-related graphs but being in two different chains which share at least one graph are called potentially \( O \)-related graphs.

We have defined four functions which offer four observable behaviours of a MUS graph \( g \). Two of them, complete traces, denoted as \( TC(g) \), and complete and non finite traces, denoted as \( TC^\infty(g) \), offer semantic viewpoints of the graph. Although both of them are based on traditional complete trace semantics \([1]\), they also take into account both true and false events in order to differentiate false events from unspecified ones. In figure below, the results of applying this two functions to a MUS graph \( g \) are shown.

As it is shown in this example, the MUS graph \( g \) has five different evolution ways, that is, it can evolve from initial state to a final one through event \( b \) followed by a state where event \( d \) is non possible; from initial state to a final one through events \( a \) and \( c \); from initial state to a final one through events \( a \), \( e \), and \( c \); from initial state to a final one through event \( a \), a number non determined of events \( e \) and, finally, event \( c \); and from initial state through event \( a \) and a number infinite of events \( e \). These sequences of possible and non possible events are stored in \( TC^\infty(g) \) and \( TC(g) \). We have to note that although trace \( ace \) is included in \( a(e) + c \) trace, both of them are explicitly included in \( TC^\infty(g) \) and \( TC(g) \) because of efficient reasons in collating tasks. The main difference between the two observation criteria is the treatment of non finite evolution ways of the graph, in \( TC^\infty(g) \) non finite traces are included—contributions \( a(e) + c \) and \( a(e) + c \) but in \( TC(g) \) they are considered as being finite ones—contributions \( ace \) and \( ae \) respectively.

The other two identified functions, number of evolutions, denoted as \( NE(g) \), and number of non finite evolutions, denoted as \( NE^\infty(g) \), offer structural viewpoints of the graph. In figure below, the results of applying this two functions to a MUS graph \( g \) are shown.

\[ NE(g) = (1, 2, 3, 3) \]
\[ NE^\infty(g) = (1, 2, 3, (2)+,(3)+) \]

\( NE(g) \) and \( NE^\infty(g) \) obtain the number of transitions that the system have to make whenever it evolves through every single evolution way. For instance when the system evolve from initial state to a final one through events \( a \) and \( c \), it implies the system has to make two transitions, being this number one of the contributions to \( NE(g) \) and \( NE^\infty(g) \). and, when the system evolves from initial state to a final one through event \( a \), a number non determined of events \( e \) and, finally, event \( c \), it implies the system has to make at least three transitions, but they could be more because of the repetition of event \( e \), which is expressed by \((3)+\) in \( NE^\infty(g) \). The main difference between them is the treatment of non finite evolution ways of the graph, in \( NE^\infty(g) \) non finite traces are included—contributions \((3)+\) from \( a(e) + c \) trace and \((2)+\) from \( a(e) + c \) trace—but in \( NE(g) \) they are considered as being finite ones—contributions \( 3 \) and \((2)+\) respectively.

These four results, \( TC^\infty(g) \), \( TC(g) \), \( NE^\infty(g) \), and \( NE(g) \), are automatically obtained from the MUS graph and they constitute the profile or set of characterizing attributes of a reusable component.

From each of these four functions—\( TC^\infty, TC, NE^\infty, NE \)—a partial order \((\subset TC^\infty, \subset TC, \subset NE^\infty, \subset NE)\) and an equivalence relation \((\equiv TC^\infty, \equiv TC, \equiv NE^\infty, \equiv NE)\) among MUS graph are defined. These relationships among reusable components allow organizing the repository in four different lattices, one for each partial ordering. As a result of this, each reusable component \((C)\) is classified in the repository after finding its correct place in each lattice. That is, it is necessary looking for those components \( O \)-related\(^3\) to \( C \) such as \( C \) is \( O \)-included on them, and those components

\(^2\)Two graphs \( g \) and \( g' \) are \( O \)-related iff \( g \subset O g' \) or \( g' \subset O g \).

\(^3\)Where \( O \in \{ TC^\infty, TC, NE^\infty, NE \} \).

\(^4\)Two components \( C \) and \( C' \) are \( O \)-related \( (C \equiv O C' \) or \( C' \equiv O C) \) iff their MUS graphs \( g \) and \( g' \) are \( O \)-related.
$O$-related to $C$ such as they are $O$-included on $C$. In order to eliminate superfluous reusable components connections, anti-symmetric property\(^5\) of $O$ is applied. An example of one of this lattice is shown in figure 2, where several reusable components are ordered according to $NE$ criteria.

6. Retrieval process

In order to reuse already developed MUS prototypes which are functionally close to the functionality specified by the query, we propose replacing the box labeled as $MUS$ in figure 1 by the sequence of tasks shown in figure 3. The retrieval process starts whenever the user expresses the required functionality by a set of SCTL requirements. From this set of SCTL requirements they are automatically obtained a set of search patterns which are used to retrieve the most suitable component from the repository. These search patterns reflect the functional characteristics expressed by the query and they are the result of applying $TC^\infty$, $TC$, $NE^\infty$ and $NE$ functions to the set of SCTL requirements [4].

![Figure 3. Retrieval process in ARIFS tool.](image)

The first step of this retrieval process is set up in two directions: on the one hand, assessing structural similarities, that is, if reusable components have similar representation to the query’s specification; and, on the other hand, assessing semantic similarities, that is, if reusable components have similar functional features to the query’s specification. As difference to previous works [9] which manage reusable components of code, in our case structural information is really close to semantic one. At this level of abstraction, a direct relationship between them is supported because MUS prototypes are automatically obtained from the set of SCTL requirements, therefore, it is impossible having two structurally different prototypes from the same functional requirements.

---

\(^5\) $C_1 \sqsubseteq O C_2$ and $C_2 \sqsubseteq O C_3$, implies $C_1 \sqsubseteq O C_3$. 

---
in this section, both rough search (section 6.1) and refined one (section 6.2) are detailed in order to retrieve from the repository the most suitable reusable component to satisfy the functionality specified by the query taking into account their semantic characteristics.

6.1. First phase: rough search

In this phase, we take advantage of static classification schemes of the repository to recover, in a quick way, a set of components functionally close to the query. This functional proximity is assessed according to structural and semantic similarities. The underlying idea of this rough search is that the closer two reusable components are, the more functional similarities they have. Four classification distances are used in this stage, one for each partial ordering criteria defined in section 5.

Definition 6.1. Let \( g \) and \( g' \) be two potentially \( O \)-related MUS graphs. Then, the classification distance between \( g \) and \( g' \), denoted by \( d_l(g, g') \), is the number of transitions which is necessary to skip on chains from the first one to the second one.

The classification distance between two non potentially \( O \)-related MUS graph is defined as \( d_l(g, g') = \infty \).

Definition 6.2. Let \( C \) be a reusable component which has been classified in a lattice according to an \( O \) criteria. The set of predecessor components of \( C \), denoted by \( \mathcal{P}(C) = \{C_i\}_{i=1}^{m} \), is such as \( C_i \sqsubseteq C \) and \( d_l(C_i, C) = 1 \); the set of successor components of \( C \), denoted by \( \mathcal{S}(C) = \{C_i\}_{i=1}^{m} \), is such as \( C \sqsubseteq C_i \) and \( d_l(C, C_i) = 1 \); and, finally, the set of equivalent components of \( C \), denoted by \( \mathcal{E}(C) = \{C_i\}_{i=1}^{m} \), is such as \( C = \sqsubseteq C_i \).

Semantic similarities predominate in \( TC \) and \( TC^\infty \) criteria, but, meanwhile, the first one does not take into account behaviour loops, the second one, on the contrary, considers them. As consequence, applying one criteria or the other only depends on if the query in turn specifies behaviour loops. So, if the query has behaviour loops, semantic retrieval is based on \( TC^\infty \) criteria, and, in other case, semantic retrieval is based on \( TC \) criteria. This decision is caused by two reasons:

- If the query specifies behaviour loops, \( TC^\infty \) criteria let us recover reusable components with a high semantic similarity to the given functionality. \( TC \) relationship does not offer more accurate results and, in this way, we are able to improve retrieval efficiency without penalizing final results.
- If the query does not specify any behaviour loop, \( TC^\infty \) does not offer a better information about reusable components than \( TC \). As comparing reusable components according to \( TC^\infty \) is more complex, we are saving resources without affecting the quality of the results.

In order to retrieve those reusable components which are close to the query, semantically speaking, ARIFS retrieves the set of successor, predecessor and equivalent components of the functionality expressed by the query, according to \( TC \) or \( TC^\infty \) criteria.

6.2. Second phase: accurate retrieval

In this second phase, we start from two different set of reusable components: those which are structurally close to the query, according to \( NE \) or \( NE^\infty \) criteria; and those which are semantically close to the query, according to \( TC \) or \( TC^\infty \) criteria. Both sets are obtained from an approximate assessment of the functional similarities and, in this second phase, the search is refined to select the most similar reusable component to the functionality expressed by the query. Each set has a different selection process, although both of them are consequence from a common idea: minimizing the adaptation efforts to the query.

Semantic search process tries to find those reusable components whose functional features are close to the given by the query. It is supposed that the components recovered in the first stage are semantically closer to the query than the remaining components in the repository, but the criteria used in the first phase to recover them from the repository only allows maintaining precedence relationships. Therefore, we have probably recovered reusable components which have notable semantic differences in spite of keeping the same partial relationship to the query. The main goal of this second phase is refining the search and assessing how are the semantic differences between these components and the query, that is, predicting the adaptation efforts to satisfy the functionality of the query; and, consequently, deciding which of them is the most suitable reusable component.

The refined retrieve according to semantic closeness to the query takes into account the events which enable the transitions of the prototypes, because this information provides the semantic of the component, that is, its meaning. In order to quantify similarities and differences among reusable components, we have defined two basic functions: functional meet and functional difference.

Definition 6.3. Given two MUS graphs \( g \) and \( g' \), the functional meet between them, denoted by \( g \land g' \), is obtained as the intersection between \( TC(g) \) and \( TC(g') \) or \( TC^\infty (g) \) and \( TC^\infty (g') \). If \( g \) or \( g' \) specifies some functional loop, then \( g \land g' = TC^\infty (g) \cap TC^\infty (g') \), in other case, \( g \land g' = TC(g) \cap TC(g') \).

\( g \land g' \) can be interpreted as the functional specifications (sequences of events) which are shared by both graphs and
it constitutes a good starting point to assess functional similarities between two reusable components.

**Definition 6.4.** Given two MUS graphs \( g \) and \( g' \), such as at least one of them specifies some functional loop, and satisfying \( g \sqsubseteq_{TC} g' \), the functional difference between them, denoted by \( g' \ominus g \), consists of those components of \( TC^\infty (g') \) which are not included in \( TC^\infty (g) \).

Given two MUS graphs \( g \) and \( g' \), such as no one of them specifies some functional loop and satisfying \( g \sqsubseteq_{TC} g' \), the functional difference between them, denoted by \( g' \ominus g \), consists of those components of \( TC^\infty (g') \) which are not included in \( TC(g) \).

\( g' \ominus g \) can be interpreted as the sequences of events which have to be added to \( g \) to make the functional specification of \( g \) the same as the functional specification of \( g' \).

These two functions (definitions 6.3 and 6.4) are the basis to define five semantic refinement functions (Table 1) which allow us to quantify and predict functional differences and similarities among graphs and, consequently, among reusable components.

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Func. consensus</td>
<td>( \rho(C, C') )</td>
<td>( g' \ominus g )</td>
</tr>
<tr>
<td>Func. deficit</td>
<td>( \delta(C, C') )</td>
<td>( g \ominus (g \ominus g') )</td>
</tr>
<tr>
<td>Func. excess</td>
<td>( \varepsilon(C, C') )</td>
<td>( g' \ominus (g \ominus g') )</td>
</tr>
<tr>
<td>Func. adaptation</td>
<td>( \Delta(C, C') )</td>
<td>( \delta(C, C') \cup \varepsilon(C, C') )</td>
</tr>
<tr>
<td>Func. adjusting</td>
<td>( \Theta(C, C') )</td>
<td>( \rho(C, C'), \Delta(C, C') )</td>
</tr>
</tbody>
</table>

Table 1. Semantic refinement functions

### 6.2.1 Functional consensus

Let \( C \) and \( C' \) be two reusable components, the functional consensus between them, denoted by \( \rho(C, C') \), is defined as \( \rho(C, C') = g' \ominus g' \), where \( g \) is the MUS graph linked to \( C \) and \( g' \) the MUS graph linked to \( C' \). As \( \rho(Q, C) \) can be interpreted as the common functionality between the current query, \( Q \), and a component \( C \), we use this functionality measurement to obtain the component \( C_i \) which maximizes \( \rho(Q, C_i) \).

Given two reusable components \( C_1 \) and \( C_2 \), the first one is closer to \( Q \) than the second one, according to functional consensus, denoted by \( \rho(Q, C_2) \sqsubseteq \rho(Q, C_1) \), whenever \( \rho(Q, C_2) \sqsubseteq \rho(Q, C_1) \).

### 6.2.2 Functional deficit

Let \( C \) and \( C' \) be two reusable components, the functional deficit of \( C' \) regarding \( C \), denoted by \( \delta(C, C') \), is defined as \( \delta(C, C') = g_1 \ominus \rho(C, C') = g \ominus (g \ominus g') \), where \( g \) is the MUS graph linked to \( C \) and \( g' \) the MUS graph linked to \( C' \). \( \delta(Q, C_i) \) expresses the functional characteristics required by \( Q \), the current query, which are not specified by \( C_i \), that is, those lack of functional requirements of \( C_i \).

Given two reusable components \( C_1 \) and \( C_2 \), the first one is closer to \( Q \) according to functional deficit criteria, denoted by \( \delta(Q, C_1) \sqsubseteq \delta(Q, C_2) \), whenever \( \delta(Q, C_1) \sqsubseteq \delta(Q, C_2) \).

### 6.2.3 Functional excess

Let \( C \) and \( C' \) be two reusable components, the functional excess of \( C' \) regarding \( C \), denoted by \( \varepsilon(C, C') \), is defined as \( \varepsilon(C, C') = g \ominus \rho(C, C') = g' \ominus (g \ominus g') \), where \( g \) is the MUS graph linked to \( C \) and \( g' \) the MUS graph linked to \( C' \). \( \varepsilon(Q, C_i) \) expresses the functional characteristics specified by \( C_i \) that are not required by the query \( Q \), that is, those extra functional specifications of \( C_i \).

Given two reusable components \( C_1 \) and \( C_2 \), the first one is closer to \( Q \) according to functional excess criteria, denoted by \( \varepsilon(Q, C_1) \sqsubseteq \varepsilon(Q, C_2) \), whenever \( \varepsilon(Q, C_1) \sqsubseteq \varepsilon(Q, C_2) \).

### 6.2.4 Functional adaptation

Let \( C \) and \( C' \) be two reusable components, the functional adaptation between them, denoted by \( \Delta(C, C') \), is defined as \( \Delta(C, C') = \delta(C, C') \cup \varepsilon(C, C') \), where \( g \) is the MUS graph linked to \( C \) and \( g' \) the MUS graph linked to \( C' \). \( \Delta(Q, C_i) \) expresses the functional characteristics being specified by the query \( Q \) not specified by \( C_i \) and vice versa, that is, those functionality which must be added or eliminated in \( C_i \) to make the functionality of \( C_i \) the same to \( C \).

Given two reusable components \( C_1 \) and \( C_2 \), the first one is closer to \( Q \) according to functional adaptation, denoted by \( \Delta(Q, C_1) \sqsubseteq \Delta(Q, C_2) \), whenever \( \Delta(Q, C_1) \sqsubseteq \Delta(Q, C_2) \).

### 6.2.5 Functional adjusting vector

We are interested in retrieving the component \( C \) which maximizes the functional consensus regarding the current query \( Q \), and minimizes its functional adaptation. The functional adjusting vector, denoted by \( \Theta(Q, C) \), takes into account both characteristics: \( \Theta(Q, C) = (\rho(Q, C), \Delta(Q, C)) \).

---

6The relationship \( \bigvee \) will be \( TC^\infty \) or \( TC \) depending on if the query...
Given two reusable components \(C_1\) and \(C_2\), the adaptation efforts of \(C_1\) to make its functional characteristics equal to the current query \(Q\) ones are less than the adaptation efforts of \(C_2\), denoted by \(\Theta(Q, C_1) \subseteq \Theta(Q, C_2)\), whenever:

\[
(\Delta(Q, C_1) \subseteq \Delta(Q, C_2))
\]

\[\lor ((\Delta(Q, C_1)) = \sigma \Delta(Q, C_2)) \land (\rho(Q, C_1) \subseteq \rho(Q, C_1))\]

6.2.6. Dynamic ordering according to the query

Given any refinement relation \(\xi\) (defined in the table 1), it is possible reordering a set of reusable components \(C = \{C_i\}_{i=1}^m\) regarding the current query \(Q\), that is, ordering the results \(\{\xi(Q, C_i)\}_{i=1}^m\) according to one of the partial ordering relation defined in 6.2.1, 6.2.2, 6.2.3, 6.2.4 or 6.2.5.

The graphical representation of a partial ordering relation can be simplified as follows: since the relation is understood to be reflexive, we can omit arrows from points back to themselves; since the relation is understood to be transitive, we can omit arrows between points that are connected by sequences of arrows; and when the representation is so oriented that all arrowheads point in one direction, we can omit the arrowheads. A graphical representation of a partial ordering relation in which all arrowheads are understood to be pointing upward is also known as the Hasse diagram of the relation [11].

![Figure 4. Example of a Hasse diagram](image)

As a consequence, these new partial ordering relations can be represented by a Hasse diagram like, for instance, in figure 4, where \(\xi(Q, C_1)\) and \(\xi(Q, C_6)\) are the superior values in the partial ordering relation defined by \(\xi\), and \(\xi(Q, C_7)\) the inferior one.

As it was explained in the previous subsections, these partial ordering relations can be reduced to a partial ordering relation defined by \(TC^\infty\) function, whenever the query \(Q\) has some functional loop, or \(TC\) function, in other case. Because of these partial ordering relations, we can find a situation where there is not any relation among the retrieved components of the first phase of the retrieval process, so in this case, we must apply a total ordering relation among them to be able to decide among the potentially suitable components:

**Definition 6.5.** Given a semantic refinement relation \(\xi\) of table 1, and given two reusable components \(C_1\) and \(C_2\) such as \(\xi(Q, C_1)\) and \(\xi(Q, C_2)\) are not \(TC\)- or \(TC^\infty\)-related (according to the characteristics of \(Q\)), the first one is closer to the query \(Q\), according to \(\xi\), criteria, than the second one, iff \(z(\xi(Q, C_1)) \leq z(\xi(Q, C_2))\). The cardinal function \(z(\xi(Q, C_1))\) obtains the number of elements of \(\xi(Q, C_1)\).

7. Example of application

In this section, we show, by using a little example, all the steps of the retrieval process set up by ARIFS tool which are involved in the semantic search. Starting from a set of SCTL requirements which constitutes a query \(Q\), we try to retrieve a reusable component from a given repository which minimizes the adaptation efforts, in order to avoid synthesis tasks and future formal verification ones. In this example, this set of SCTL requirements is as follows:

\[
Q \equiv \ (a \Rightarrow \bigcirc R_1) \Rightarrow (b \Rightarrow \bigcirc d) \\
R_1 \equiv (c \Rightarrow (e \Rightarrow \bigcirc R_1))
\]

The first step of the retrieval process starts obtaining the search patterns of the query. As \(Q\) has functional loops, the search pattern is expressed in terms of \(TC^\infty\) function: \(TC^\infty(Q) = (a(e)+, ac, ace, a(e)+c, bd)\), and in terms of \(NE^\infty\) one: \(NE^\infty(Q) = ((2)+, 2, 3, (3)+, 2)\). As in this paper, we focus on the semantic viewpoint of the retrieval process, the following information is only relating to this characteristic of the retrieval process.

![Figure 5. Repository example](image)

Assuming that the repository consists of the six reusable components whose MUS graphs are shown in figure 5, the
result of the rough search, according to $TC^\infty$ criteria, are both the set of predecessor reusable components $P(Q) = \{ g_1, g_6 \}$ and the set of successor ones $S(Q) = \{ g_6 \}$. Therefore, the set $C = P(Q) \cup S(Q)$ constitutes the starting point of the second phase of the retrieval process, that is, the refined retrieval.

As we try to recover the component which minimizes the adaptation efforts to the query, the refined criteria must be the functional adjusting vector. Firstly, we obtain the $TC^\infty$ function applied to every reusable component in $C$:

\[
TC^\infty(g_1) = (ac, bd) \\
TC^\infty(g_6) = (a(e) + , bd) \\
TC^\infty(g_6) = (ac, aec, a(e) + , a(e) + c, e(f) + , bd)
\]

and, secondly, the vector $\Theta(Q, g_i)$ must be obtained for each one of these reusable components of $C$. The following tables store the intermediate steps to calculate these vectors and the final results.

<table>
<thead>
<tr>
<th>$g_i$</th>
<th>$\rho(Q, g_i)$</th>
<th>$\epsilon(Q, g_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_1$</td>
<td>$(ac, bd)$</td>
<td>{}</td>
</tr>
<tr>
<td>$g_6$</td>
<td>$(ac, aec, a(e) + , a(e) + c, bd)$</td>
<td>$(e(f) +)$</td>
</tr>
<tr>
<td>$g_5$</td>
<td>$(a(e) + , bd)$</td>
<td>{}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$g_i$</th>
<th>$\delta(Q, g_i)$</th>
<th>$\Delta(Q, g_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_1$</td>
<td>$(a(e) + , aec, a(e) + c)$</td>
<td>$(a(e) + , aec, a(e) + c)$</td>
</tr>
<tr>
<td>$g_6$</td>
<td>{}</td>
<td>$(e(f) +)$</td>
</tr>
<tr>
<td>$g_5$</td>
<td>$(ac, aec, a(e) + c)$</td>
<td>$(ac, aec, a(e) + c)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$g_i$</th>
<th>$\Theta(Q, g_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_1$</td>
<td>$((ac, bd), (a(e) + , aec, a(e) + c))$</td>
</tr>
<tr>
<td>$g_6$</td>
<td>$((ac, aec, a(e) + , a(e) + c, bd), (e(f) +))$</td>
</tr>
<tr>
<td>$g_5$</td>
<td>$((a(e) + , bd), (ac, aec, a(e) + c))$</td>
</tr>
</tbody>
</table>

After obtaining the vectors $\{ \Theta(Q, g_i) \ \forall g_i \in C \}$, we apply the partial ordering relationship defined in section 6.2.5 whose result is shown in figure 6(a). As with this partial ordering we are not able to conclude which of the reusable components needs less modifications to be adapted to the current query, the total order among them — according to the definition 6.5— is obtained (figure 6(b)). Therefore, $g_6$ is the reusable component which needs less adaptation efforts to $Q$.

8. Selection and adaptation

Once the retrieval process has finished, the user has to select between two reusable components: the closest reusable component to the current query, from a structural viewpoint; or the closest reusable component to the query, from a semantic viewpoint. This decision must be done according to the adaptation efforts offered and predicted by ARIFS in both cases —structural distances or semantic differences.

Whenever the selected reusable component is the most semantically similar to the query, semantic adaptation tasks are needed. Firstly, increasing the functionality specified by the retrieved component $g$ is needed in order to satisfy the functionality specified by the query $Q$. ARIFS sets up this adaptation task starting from $\delta(Q, g)$, which predicts this increasing effort. Secondly, it is necessary to eliminate the functionality specified by the retrieved component $g$ which is not specified by the query $Q$. This reduction task is predicted by $\epsilon(Q, g)$ which is used by ARIFS to adapt $g$. Finally, the verification information linked to $g$ is modified to maintain these verification results in effect after the adaptation process [6].

In the example of the previous section, we have retrieved the reusable component whose MUS graph is $g_6$ (figure 7(a)) and it is adapted to satisfy the requirements expressed by the query $Q$: after eliminating the traces in $\epsilon(Q, g_6) = (e(f) +)$, the prototype obtained is shown in figure 7(b), which satisfies the requirements expressed by the query $Q$.
9. Summary and future work

The work introduced in this paper focuses on reusing incomplete prototypes to reduce synthesis tasks in the requirements specification stage of a totally formalized, incremental and iterative software development process. Not only synthesis tasks are reduced, but also formal verification ones, because each reusable component also stores previous formal verification results [6]. Therefore, software components from the requirements specification phase are reused, which although being more complex is also more interesting than reusing code components.

We propose a classification mechanism based on four different partial orderings among incomplete prototypes, and a two-tiered approximate retrieval where formal proofs are avoided. The first phase takes advantage of the static classification to recover a small set of potentially suitable reusable components. In the second phase, several refinement relations are used to predict adaptation efforts (dynamic ordering), and, consequently, to select the closest reusable component to the functionality specified by the query. Although both structural and semantic differences are assessed, in this paper, we have only focused on semantic differences as a way to reuse MUS prototypes within same domain or application area, which is known as vertical reuse.

In order to continue this proposal, we are working on the selection phase. Nowadays, the user has to decide between two reusable components, the closest one to the query according to structural similarities and the closest one to the query according to semantic similarities. At this point, the user only has the prediction of adaptation efforts, but these adaptation efforts are assessed in different terms. We propose unifying the adaptation efforts assessing the semantic adaptation tasks and the structural ones to provide this information to the user.

References


