Approximate Retrieval of Incomplete and Formal Specifications applied to Horizontal 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

A key problem in software reuse is the selection of appropriate components for satisfying a given requirement, being this problem more pronounced in horizontal reuse, that is, the reuse of software elements in different domains of applications. 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 applying formal verification tasks, and to predict adaptation efforts to satisfy the required functional specification. This paper focuses on structural similarities because of their flexibility to adapt requirements specifications from different domains.

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 [9]. 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 functional relationships among reusable components; the retrieval process is explained in section 6, including the defined measures to quantify functional differences; in section 7 the selection and adaptation of reusable components is explained; and, finally, a brief summary and future work are exposed in section 8.

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 [13] the retrieval process of code components is based on the Larch/ML specification language for component description and the associated interactive Larch prover for re-
treival. Formal proofs are restricted to a small subset of the repository, which is previously selected by using preconditions and postconditions. REBOUND (REuse Based On UNDERTstanding) tool [11] 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 [3] a two-tiered hierarchy of the repository based on formal specifications using OSPL is proposed. The lower level is based on general relationship; 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 [12] 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. As applying theorem proving in the retrieval process is very difficult to automate, Fisher proposes [7] library browsing —using a special navigator—as an alternative to library retrieval.

3. Context

In this section we briefly describe the software development process, SCTL-MUS methodology [10], 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 [10] (Simple Causal Temporal Logic)—in box labeled as SCTL in figure 1. A generic causal requirement in SCTL follows this pattern:

\[ \text{Premise} \Rightarrow \Box \text{Consequence}, \]

which establishes a causing condition (premise); a temporal operator determining the applicability of the cause (\( \Rightarrow \Box \)); 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 (\( \frac{1}{2} \)). 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.

![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.

![Example of MUS graph.](image)

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 $a_{unspec}$. 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 \(^1\).

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 into 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 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 $\mathcal{O}$ that associates with every MUS graph $g$ a set $\mathcal{O}(g)$ which constitutes the observable behaviour of $g$. For every such $\mathcal{O}$, the equivalence relation $\equiv_{\mathcal{O}} \subseteq \mathcal{G} \times \mathcal{G}$ is given by $g \equiv_{\mathcal{O}} g' \iff \mathcal{O}(g) = \mathcal{O}(g')$, and the preorder $\sqsubseteq_{\mathcal{O}} \subseteq \mathcal{G} \times \mathcal{G}$ by $g \sqsubseteq_{\mathcal{O}} g' \iff \mathcal{O}(g) \subseteq \mathcal{O}(g')$, that is, $\sqsubseteq_{\mathcal{O}}$ provides a partial order between equivalence classes or graph sets indistinguishable using $\mathcal{O}$-observations, so $(\mathcal{G}, \sqsubseteq_{\mathcal{O}})$ is a partially ordered set, or poset. A subset $G_1 \subseteq \mathcal{G}$ is called a chain if every two graphs in $G_1$ are $\mathcal{O}$-related\(^2\). Two graphs non $\mathcal{O}$-related but being in two different chains which share at least one graph are called potentially $\mathcal{O}$-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

---

\(^1\)Unspecified events of a state are not represented, only $a_{unspec}$ of $E_0$ because it implies an evolution of the model.

\(^2\)Two graphs $g$ and $g'$ are $\mathcal{O}$-related iff $g \sqsubseteq_{\mathcal{O}} g'$ or $g' \sqsubseteq_{\mathcal{O}} g$. 

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.

\[
\begin{align*}
T C(g) &= (b \land \neg d, a \lor e, a \lor e, a \lor e, a \lor e) \\
\end{align*}
\]

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 \( e \) and finally, event \( c \); from initial state through event \( a \) and a number infinite of events \( e \). These sequences of possible and non possible events are stored in \( T C^\infty(g) \) and \( T C(g) \). We have to note that although trace \( a \lor e \) is included in \( a \lor c \) trace, both of them are explicitly included in \( T C^\infty(g) \) and \( T C(g) \) 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 \( T C^\infty(g) \) non finite traces are included — contributions \( a \lor e + c \) and \( a(e) + \ldots \) but in \( T C(g) \) they are considered as being finite ones — contributions \( a \lor e \) and \( a \lor e \) respectively.

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

\[
\begin{align*}
N E(g) &= (1, 2, 2, 3, 3) \\
N E^\infty(g) &= (1, 2, 3, (2)^+, (3)^+) \\
\end{align*}
\]

\( N E(g) \) and \( N E^\infty(g) \) 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 \( N E(g) \) and \( N E^\infty(g) \). 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 \( N E^\infty(g) \). The main difference between them is the treatment of non finite evolution ways of the graph, in \( N E^\infty(g) \) non finite traces are included — contributions \((3)^+ \) from \( a(e) + c \) trace, and \((2)^+ \) from \( a(e) + c \) trace— but in \( N E(g) \) they are considered as being finite ones — contributions 3 and 2 respectively.

These four results, \( T C^\infty(g), T C(g), N E^\infty(g), \) and \( N E(g) \), 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 — \( T C^\infty, T C, N E^\infty, \) and \( N E \) — a partial order (\( \sqsubseteq TC^\infty, \sqsubseteq TC, \sqsubseteq NE^\infty, \sqsubseteq NE \)) and an equivalence relation (\( = TC^\infty, = TC, = NE^\infty, = NE \)) among MUS graphs 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 \( O \)-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.

Applying the four defined relationships among graphs, we are able to distinguish between semantic similarities, which reflects if two reusable component have similar functional features (considering sequences of events), that is, if they act alike; and structural similarities, which reflects if two reusable component have similar representation of their MUS graph without considering events linked to each transition (only the skeleton of the graph), that is, if they look alike. Semantic closeness provides a suitable frame to vertical reuse, that is, the reuse of specifications within the same domain or application area. Components of these applications may partially share the alphabet of events of the graphs. Structural closeness, on the other hand, provides a suitable frame to horizontal reuse, that is, the reuse of specifications in different applications, so components of these applications hardly share the alphabet of events and it would be necessary to do mapping tasks (section 7). As difference to previous works [8] 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 STCL requirements, therefore, it is impossible having two structurally

---

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

\(^4\)Two components \( C \) and \( C' \) are \( O \)-related (\( C \sqsubseteq O C' \) or \( C' \sqsubseteq O C \)) iff their MUS graphs \( g \) and \( g' \) are \( O \)-related.

\(^5\)\( C_1 \sqsubseteq C_2 \) and \( C_2 \sqsubseteq C_3 \), implies \( C_1 \sqsubseteq C_3 \).
different prototypes from the same functional requirements.

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 /CC/BV, /CC/BV, /C6/BX, and /C6/BX functions to the set of SCTL requirements [4].

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. In this paper, we focus on evaluation of structural similarities among reusable components and the query, therefore 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 structural 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 classify in the repository, 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 \( d_c(g, g') \), 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 \( d_c(g, g') = \infty \).

**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 \( P(C) = \{ C_i \}_{i=1}^{n} \), is such as \( C_i \sqsubseteq_o C \) and \( d_c(C_i, C) = 1 \); the set of successor components of \( C \), denoted by \( S(C) = \{ C_i \}_{i=1}^{m} \),...
is such as $C \subseteq \sigma C_i$ and $d_l(C, C_i) = 1$; and, finally, the set of equivalent components of $C$, denoted by $\mathcal{E}(C) = \{ C_i \}_{i=1}^k$, is such as $C = \sigma C_i$.

Structural similarities predominate in $NE$ and $NE^\infty$ 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, structural retrieval is based on $NE^\infty$ criteria, and, in other case, structural retrieval is based on $NE$ criteria. This decision is caused by two reasons:
- If the query specifies behaviour loops, $NE^\infty$ criteria let us recover reusable components with a high structural similarity to the given functionality. $NE$ 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, $NE^\infty$ does not offer a better information about reusable components than $NE$. As comparing reusable components according to $NE^\infty$ 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, structurally speaking, ARIFS retrieves the set of successor, predecessor and equivalent components of the functionality expressed by the query, according to $NE^\infty$ or $NE$ 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.

Structural search process tries to find those reusable component whose skeleton are close to the given by the query. It is supposed that the components recovered in the first stage are structurally closer to the query than the remaining components in the repository, but the criteria used to recover them from the repository only allows us to maintain precedence relationships. Therefore, we have probably recovered reusable components which have notable structural 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 is the difference 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 going to be the most suitable reusable component. In order to be able to quantify this structural differences among components, we define two different measures of structural differences: number of evolutions distance (section 6.2.1) which is based on $NE$ criteria; and number of non finite evolutions distance (section 6.2.2) which is based on $NE^\infty$ criteria. As we explained in the previous section, the decision of applying one or the other one only depends on the characteristics of the query, if the query has behaviour loops the number of evolutions distance is used, in other case, the number of non finite evolutions distance is used.

#### 6.2.1. Number of evolutions distance

Number of evolutions distance allows quantifying structural differences between two reusable components according to the $NE$ criteria.

**Definition 6.3.** Let $g$ and $g'$ be two MUS graphs, the number of evolution distance between them, denoted by $d_{NE}(g, g')$, is the Euclidean distance between two vectors that being contained in $NE(g)$ and $NE^\infty(g)$ respectively, they do not have any element in common, they have the same number of elements (in other case the smallest vector is completed with zeros), and their elements are sorted according to increasing order.

![Figure 4. Example of $NE$ ordering.](image)

In figure 4 an example of lattice according to $NE$ criteria is shown. In this case, we can see that both $g_3$ and $g_7$ are the predecessor and successor respectively of the graph $g_6$, so they satisfy $d_l(g_3, g_6) = d_l(g_6, g_7) = 1$, but $g_3$ and $g_7$
are clearly different. In order to know which of them is
the closest one to \( g_6 \), according to \( NE \) criteria, we use the
number of evolutions distance. Firstly, the \( NE \) function is
applied to these three graphs:

\[
NE(g_3) = (2, 1, 2, 2) \quad NE(g_6) = (2, 2, 2, 3, 3) \quad NE(g_7) = (2, 2, 2, 3, 3)
\]

and, secondly, we obtain the number of evolutions distance
between \( g_3 \) and \( g_6 \) and between \( g_7 \) and \( g_6 \):

\[
d_{\text{NE}}(g_3, g_6) = d_{\text{NE}}((2, 1, 2, 2), (2, 2, 3, 3)) = \| (1, 2) - (3, 3) \| = \sqrt{5}
\]

\[
d_{\text{NE}}(g_7, g_6) = d_{\text{NE}}((2, 2, 2, 3, 3), (2, 2, 2, 3, 3)) = \| (2) - (0) \| = 2
\]

therefore, as \( d_{\text{NE}}(g_3, g_6) < d_{\text{NE}}(g_7, g_6) \), \( g_7 \) is closer to \( g_6 \)
than \( g_3 \) according to \( NE \) criteria.

### 6.2.2. Number of non finite evolutions distance

The possibility of loops is not considered in number of evolutions
distance (definition 6.3), so in order to take into
account the functional loops specified in MUS graphs, another
measure of structural differences is needed. **Number of non finite evolutions distance** is based on \( NE^\infty \) criteria, so it takes into account this characteristic. \( NE^\infty (g) \) can be studied as the composition of two different vectors:
the first one, \( NE^\infty_1 (g) \), is the result of applying \( NE^\infty \) criteria to every single evolution path specified in the graph
which does not present any functionality loop; and the second one, \( NE^\infty_2 (g) \), is the result of applying \( NE^\infty \) criteria to every single evolution path specified in the graph which present some functionality loop; therefore, \( NE^\infty (g) = (NE^\infty_1 (g), NE^\infty_2 (g)) \).

**Definition 6.4.** Let \( g \) and \( g' \) be two MUS graphs and \( NE^\infty (g) \) and \( NE^\infty (g') \) the results of applying \( NE^\infty \) function to \( g \) and \( g' \) respectively. The structural difference vector, denoted by \( d_{\text{SE}}(g, g') \), is obtained as follows:

1. \( d_{\text{SE}}^{\text{r}} \) is composed of every component of \( NE^\infty_1 (g) \) and every component of \( NE^\infty_1 (g') \), after eliminating those components which are identical in both vectors. Components of \( d_{\text{SE}}^{\text{r}} \) must be sorted according to increasing ordering.

2. \( d_{\text{SE}}^{\text{r}} \) is composed of every component of \( NE^\infty_2 (g) \) and every component of \( NE^\infty_2 (g') \), after eliminating those components which are identical in both vectors. Components of \( d_{\text{SE}}^{\text{r}} \) must be sorted according to increasing ordering.

where \( d_{\text{SE}}(g, g') = (d_{\text{SE}}^{\text{r}}, d_{\text{SE}}^{\text{r}}) \).

**Definition 6.5.** Let \( g \) and \( g' \) be two MUS graphs, the number of non finite evolutions distance between them, denoted by \( d_{\text{NE}}(g, g') \), is the norm of the vector \( d_{\text{NE}}(g, g') = (d_{\text{NE}}^{\text{r}}, d_{\text{NE}}^{\text{r}}) \), where \( d_{\text{NE}}^{\text{r}} \) is obtained as in definition 6.4, and \( d_{\text{NE}}^{\text{r}} \) is obtained from \( d_{\text{NE}}^{\text{r}} \), after eliminating every single element with recursivity, \((+)\), and subtracting one unit to the result.

In figure 5 an example of lattice according to \( NE^\infty \) criteria is shown. In this case, we can see that \( \{ g_5, g_7 \} \) and \( g_7 \) are predecessors and successor graphs respectively of the graph \( g_6 \), so they satisfy \( d_{\text{NE}}(g_7, g_6) = d_{\text{NE}}(g_5, g_6) = d_{\text{NE}}(g_7, g_6) = 1, \) but they are clearly different. In order to know which of them is the closest one to \( g_6 \) according to \( NE^\infty \) criteria, we use the number of non finite evolutions distance. Firstly, the \( NE^\infty \) function is applied to these four graphs:

\[
NE^\infty (g_6) = (2, 2, 3, 2+, 3+)
\]

\[
NE^\infty (g_3) = (2, 2, 1, 2, 2)
\]

\[
NE^\infty (g_5) = (2, 2+)
\]

\[
NE^\infty (g_7) = (2, 2, 3, 2+, 2+, 3+)
\]

secondly, structural difference vectors are obtained:

\[
d_{\text{SE}}(g_5, g_6) = (2, 3, 3+)
\]

\[
d_{\text{SE}}(g_7, g_6) = (2+)
\]

\[
d_{\text{SE}}(g_3, g_6) = (1, 2, 2, 3, 2+, 3+)
\]

thirdly, we obtain \( d_{\text{SE}}^\infty \) vector between \( S(g_6) \) and \( g_6 \) and between \( P(g_6) \) and \( g_6 \); and, finally, structural differences
according to $NE^\infty$ criteria are obtained:

$$d_{NE}(g_7, g_6) = 1$$
$$d_{NE}(g_7, g_6) = \sqrt{15}$$
$$d_{NE}(g_7, g_6) = \sqrt{23}$$

so, $g_7$ is the closest graph to $g_6$, that is, the structural adaptation effort of $g_7$ to $g_6$ according to $NE^\infty$ criteria is the lowest one.

7. 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 structurally similar to the query, a mapping or renaming task is needed in order to adapt the semantic of the reusable component to the query’s one. This mapping is also extended to the verification information linked to the reusable component, that is, every single property which has been verified in the component is adapted to maintain its verification results in effect after the adaptation process.

8. 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, two different metrics are used to predict adaptation efforts (dynamic ordering), and, consequently, to select the closest reusable component to the functionality expressed by the query. Although both structural and semantic differences are assessed, in this paper, we have only described structural differences as a way to reuse these prototypes in different domains, which is known as horizontal 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