Specification Matching of Object-oriented Components

Frank Feiks
Institute for Software Engineering and Theoretical Computer Science
Technical University of Berlin
feiks@cs.tu-berlin.de

David Hemer
Software Verification Research Centre
School of Information Technology and Electrical Engineering
University of Queensland
hemer@itee.uq.edu.au

Abstract

Object-orientation supports software reuse via features such as abstraction, information hiding, polymorphism, inheritance and redefinition. However, while libraries of classes do exist, one of the challenges that still remains is to locate suitable classes and adapt them to meet the specific requirements of the software developer. Traditional approaches to library retrieval are text-based; it is therefore difficult for the developer to express their requirements in a precise and unambiguous manner. A more promising approach is specification-based retrieval, where library component interfaces and requirements are expressed using a formal specification language. In this case retrieval is based on matching formal specifications. In this paper we describe how existing approaches to specification matching can be extended to handle object-oriented components.

Keywords: *specification matching, OO components, retrieval*

1. Introduction

Object-orientation [10] is a paradigm that supports software reuse by providing mechanisms such as abstraction, information hiding, polymorphism, inheritance and redefinition. Various libraries of reusable object-oriented classes exist, e.g., LEDA [8], KARLA [14] and the libraries supplied with Eiffel [9] and Smalltalk [3], that can be used by developers to reduce development time and effort. However, while such libraries exist, the developer is still faced with the problem of locating a library class that satisfies the developer's requirements. Furthermore, having found a suitable class, the developer needs to adapt the class to meet their specific requirements.

Early approaches to retrieval were keyword based, borrowing from many of the ideas of information retrieval systems. However such retrieval methods rely on textual interfaces, which are often ambiguous, verbose and lacking in precision. A more promising approach that overcomes these problems, commonly referred to as *specification matching* [12, 7, 16], uses formal specification notations as component interfaces and bases retrieval on matching these formal specifications.

Existing specification matching approaches focus mainly on matching of functional specifications, specified in terms of pre- and post-conditions. Matching relationships are defined between a query component, Q, encapsulating the user's requirements, and a library component specification, S. The query and library component are specified using a common specification language.

A variety of matching relationships have been defined, which can be used in various situations. For example, the matching relationship *exact match* succeeds when the corresponding pre- and post-conditions of Q and S are logically equivalent. This matching technique can be used to locate a library component that can replace the corresponding query component. Relaxations of exact matching are also defined; for example *plug-in match* succeeds when the pre-condition of S is weaker than that of Q and the post-condition of S is stronger than that of Q. Such a technique can be used to find a component S that can implement the requirements of Q (as opposed to replacing Q). Another matching technique, *plug-in post*, succeeds when the post-condition of S is stronger than that of Q. Such a matching technique can be used to find a partial solution for the query Q [11].

These function-level specification matching techniques

Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 IEEE

have been extended to the module level (containing a collection of units) by defining a query to be a set of user requirements [4, 16]. Modules are specified by specifying each of the individual units in the module. A query matches a module if *all* query requirements are matched against a module unit specification. Additional algorithms have been defined for matching *some* of the query requirements, and matching *exactly one* of the query requirements [4].

Currently, specification matching is limited to matching of individual functions and simple flat modules [16, 5]. In this paper we describe how these techniques can be extended to handle matching of object-oriented classes, with particular attention paid to specific object-oriented mechanisms, such as information hiding, inheritance and redefinition.

Furthermore, in this paper we shall describe matches between a query and library class in terms of an adaptation of the library class, based on the framework described in [4]. The advantage of this is that we not only achieve computer-assisted retrieval of components, but also computer-assisted adaptation of components.

In Section 2 we introduce a basic approach to specification matching of object-oriented components, where it is assumed that the query and library classes have an equivalent data representation. In Section 3 we define a more advanced version of specification matching for classes in which coupling invariants are used to describe a relationship between the data representations of the query and library class. In Section 4 we discuss some of the practicalities associated with implementing specification matching algorithms and using these algorithms to develop a retrieval tool.

2. Basic matching

In this section we will illustrate, using an example, how existing specification matching techniques [16] can be extended to handle matching of object-oriented components. Class matching extends the notion of module matching [4], with particular emphasis on handling inheritance (we shall only consider inheritance within the library component and not in the query). In order to match classes we require techniques for matching the individual units within classes, in particular *methods* and *attributes*. Method matching extends function matching techniques, while attribute matching is based on the state schema matching routines used for matching state-based modules [5].

The example involves matching a mouse pointer query class against a 2D-vector library class. The query and library classes are specified using the Object-Z specification language [15].

2.1. Search query class

The class *MousePointer*, shown in Fig. 1, specifies the user's requirements. The first line of the class is the visibility list, indicating which attributes and methods are visible from outside of the list (referred to as *public* entities). In this case all attributes and methods are public. The class includes two attributes, *posX* and *posY*, both modelled as integers, representing the current position of a mouse pointer on the screen. An invariant is given for these attributes stating that the *X* and *Y* coordinates must stay within the range $(0\,..\,1500)$.

MousePointer

$\upharpoonright(posX, posY, move, setTo)$

$posX, posY : \mathbb{Z}$

$0 \leq posX \leq 1500$
$0 \leq posY \leq 1500$

move

$\Delta(posX, posY)$
$deltaX?, deltaY? : \mathbb{Z}$

$0 \leq posX + deltaX? \leq 1500$
$0 \leq posY + deltaY? \leq 1500$
$posX' = posX + deltaX?$
$posY' = posY + deltaY?$

setTo

$\Delta(posX, posY)$
$newX?, newY? : \mathbb{Z}$

$0 \leq newX? \leq 1500$
$0 \leq newY? \leq 1500$
$posX' = newX?$
$posY' = newY?$

Figure 1. Specification of the *MousePointer* search query class

The *MousePointer* class also includes the specification of two methods, *move* and *setTo*. The *move* method changes the position of the mouse pointer by altering the *x* and *y* coordinates by *deltaX*? and *deltaY*? respectively. (In ObjectZ, the suffix "?" is used to indicate input variables, while the suffix "!" is used to indicate output variables). The *setTo* method moves the mouse pointer to a position specified by the inputs *newX*? and *newY*?.

Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 IEEE

2.2. Library classes

A candidate match with the *MousePointer* query class is the 2*DVector* library class, which implements a two dimensional vector. The 2*DVector* class inherits from the 2*Tuple* library class shown in Fig. 2.

The 2*Tuple* library class, shown in Fig. 2, is parameterised over the types of the first and second elements of each 2-tuple (types $T1$ and $T2$ respectively). The 2*Tuple* class includes two attributes, *first* and *second*, representing the first and second elements of the tuple. The class also includes three methods: *create*; *getFirst*; and *getSecond*.

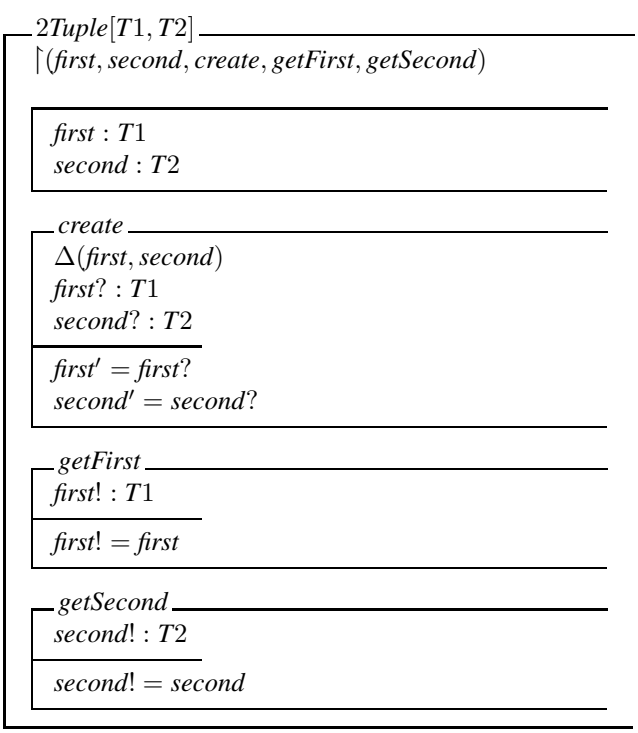

Figure 2. Specification of the 2Tuple library class

The method *create* updates the current state by assigning the input variables *first*? and *second*? to the attributes *first* and *second*.

The method *getFirst* assigns the value of the attribute *first* to the output variable *first*!. Similarly, the method *getSecond* assigns the value of the attribute *second* to the output variable *second*!. Neither of these two operations changes the state.

The visibility list of the 2*Tuple* class indicates that all attributes and methods are visible from outside of the class (i.e., they have been made *public*).

The 2*DVector* class, shown in Fig. 3, is parameterised over the type of the elements (T) in the vector. 2*DVector* inherits the 2*Tuple* class, instantiating the parameters $T1$ and $T2$ from the parent class to the parameter T. As well as inheriting the attributes and methods from its parent class, 2*DVector* includes three new methods: *add*; *scale* and *swap*.

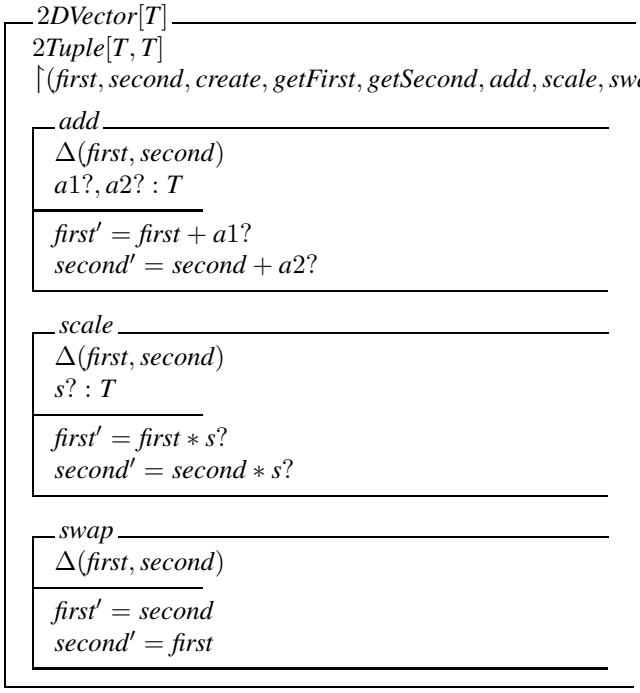

Figure 3. Specification of the *2DVector* library class

The method *add* adjusts the values of the attributes by adding the input values $a1?$ and $a2?$ to the *first* and *second* attribute respectively. The method *scale* applies a multiplier $s?$ to the attributes. The method *swap* swaps the values of the *first* and *second* attributes.

2.3. Matching the query against the library classes

Informally, we say that a query class Q matches a library class specification S, if there is some *adaptation* of the library class, such that each of the attributes and methods Q can be matched against a *visible* attribute or method from S or one of its *ancestor* classes. In the case where inherited methods are *redefined* in the child class, we only attempt to match against the method in the child class, and not against the method in the ancestor class.

One particular adaptation, π, of the the library class 2*DVector* (and the inherited class 2*Tuple*) that results in a match with *MousePointer* is:

Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 **IEEE**

1. instantiate the parameter T to $\mathbb{Z}$;
2. rename the (inherited) state variables *first* to *posX* and *second* to *posY*;
3. rename the method *add* to *move*, and rename the local variables $a1?$ and $a2?$ to *deltaX*? and *deltaY*? respectively;
4. rename the (inherited) method *create* to *setTo*, and rename the local variables *first*? and *second*? to *newX*? and *newY*? respectively.

Formally, we write π as:

$$\pi = \{T \mapsto \mathbb{Z}, \mathit{first} \mapsto \mathit{posX}, \mathit{second} \mapsto \mathit{posY}, \\ \mathit{add} \mapsto \mathit{move}, a1? \mapsto \mathit{deltax}?, a2? \mapsto \mathit{deltaY}?, \\ \mathit{create} \mapsto \mathit{setTo}, \mathit{first}? \mapsto \mathit{newX}?, \mathit{second}? \mapsto \mathit{newY}?\}$$

Attribute matching Within this basic module matching framework, we say that the attributes of the query Q and library component specification S match under an adaptation π, iff for each attribute of Q there is an equivalent attribute in $S(\pi)$. More precisely, for each attribute $\alpha : \tau_1$ in Q, there is an attribute $\beta : \tau_2$ in S or one of its ancestor classes such that β is renamed to α by π, and τ_2 is adapted to a type that is equivalent to τ_1 by π.

After matching the individual attributes, the invariants of the query Q and library specification component S are compared. This is necessary to determine whether the library component can be used in place of the query. It will also give an indication of what matching techniques can be used for matching individual methods, and how these library methods can be used in place of the corresponding query methods. The simplest case is where the invariant of Q is stronger than that of S; i.e., the methods of the library class are applicable to a wide range of values. In this case library methods can be used as-is in place of corresponding query methods. At the other extreme is when the invariant of Q is weaker than that of S; i.e., the methods of the library class are applicable to less values than those of query class. In this case the library methods can be used as a *partial* implementation for the corresponding query methods. By partial we mean that the library method satisfies the query only for particular input values. In practice we might implement the query by doing case analysis on the input values, using the library component for input values for which the library component invariant holds, and finding alternate library components for the remaining input values.

In the example, the attributes *posX* and *posY* from the query class can be matched against the attributes *first* and *second* from the 2*Tuple* class, which is inherited by the 2*DVector* class. In this case the types of the attributes are the same under the adaptation π. Under the adaptation π, the invariant of *MousePointer* is stronger than the (trivial) invariant of 2*DVector*. Therefore methods from the library class that match methods from the query can be used as-is in place of the corresponding query methods.

Method matching Method matching succeeds when each method from the query can be matched against a *visible* method from the library class or one of its *ancestor* classes. When library methods are redefined, the child method is used for matching, but not the corresponding parent method.

Individual methods are matched using techniques based on the function specification matching techniques of Zaremski and Wing [16]. However these definitions are extended to include adaptation of the library method, based on a general framework for incorporating component adaptation with specification-based matching [4]. In this paper, adaptations are assumed to consist of renamings and parameter instantiations.

For the example we restrict our attention to those matching techniques that find library methods that can be used as-is in place of the query methods, while maintaining the correctness of the overall program. Such matching techniques are generally desirable when the invariant of the query is stronger than that of the library class; in these cases calls to library methods can be plugged-in directly in place of the calls to the corresponding query method. For a formal proof that this general class of specification matching techniques maintains correctness of the overall program, the reader is referred to [2].

The *exact match* method [16, Definition 4], in which the pre-conditions and post-conditions of the query and library component specification are equivalent, is extended to include adaptation as follows:

Definition 1 *A library class specification S is said to be an* exact match with adaptation *of a query class specification Q, with respect to an adaptation π iff*

$$Q_{pre} \Leftrightarrow S_{pre}(\pi) \wedge S_{post}(\pi) \Leftrightarrow Q_{post}$$

The library method *create* from 2*DVector* can be matched against the query method *setTo* from *MousePointer* using *exact match with adaptation*, with respect to adaptation π. The matching condition is trivial in this case.

The method *add* from 2*DVector* is matched against the method *move* from the query class *MousePointer*, using an extension of *plug-in match* [16, Definition 5]. Plug-in matches query methods against library methods with weaker preconditions and stronger postconditions.

Definition 2 *A library class specification S is said to be a* plug-in match with adaptation *of a query class specification Q, with respect to an adaptation π iff*

$$(Q_{pre} \Rightarrow S_{pre}(\pi)) \wedge (S_{post}(\pi) \Rightarrow Q_{post})$$

Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 IEEE

For the methods *add* and *move* the resulting proof obligation is as follows:

$$(0 \leq posX + deltaX? \leq 1500 \land \\ 0 \leq posY + deltaY? \leq 1500) \Rightarrow \text{true} \land \\ (posX' = posX + deltaX? \land posY' = posY + deltaY?) \Rightarrow \\ (posX' = posX + deltaX? \land posY' = posY + deltaY?)$$

The proof of this condition is trivial.

3. Advanced matching

In the previous section we looked at an example where there was simple one-to-one relationship between the attributes in the query and the library component. However, typically such a simple relationship is not apparent. In this section we extend the discussion of class matching techniques from the previous section to allow more complex relationships between query and library class attributes, described in terms of a *coupling invariant* [1].

3.1. Query class

The query class *Dictionary*, shown in Fig. 4, specifies the user's requirement for our second example. The *Dictionary* class defines the *words* attribute, modelled as a set of words, used to represent the current contents of the dictionary. An initialisation method, *Init*, is given, which initialises the dictionary to the empty set. Methods are also given for: adding a word to the dictionary (*add*); deleting a word from the dictionary (*delete*); and checking whether a word exists in the dictionary (*check*).

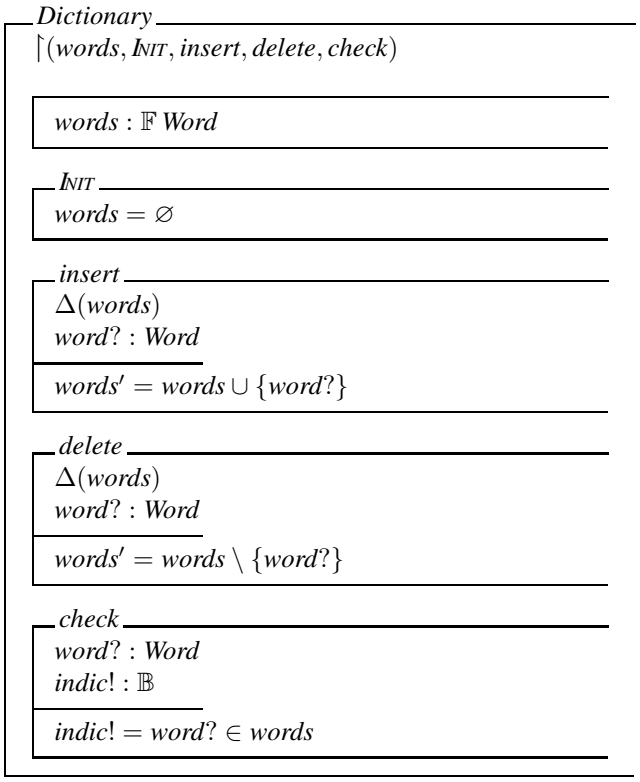

Figure 4. Specification of the *Dictionary* search query class

3.2. Library classes

The classes *Multiset* and *Set* represent existing library components. The class *Multiset*, shown in Fig. 5, specifies a multiset container class. The state is defined in terms of the current elements in the multiset (represented by the type bag). The class includes methods for: initialising the multiset; adding an element to the multiset; removing an element; and checking whether an element is contained in the current multiset.

The class *Set*, shown in Fig. 6, inherits from *Multiset* and specialises it by ensuring that no element appears more than once in the multiset. The specialisation is done by adding a class invariant and redefining the *add* operation in the child class. The invariant states that each element can appear only once (or not at all) in the set. The *add* operation is redefined by adding a precondition stating that the element to be added cannot already appear in the set.

Note that the structure of the query and library classes in this example is similar to the structure in the previous example. In particular we have a single query class, and a

Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 **IEEE**

Multiset[*T*]

$\restriction(\textit{INIT}, add, delete, member)$

$elems : \text{bag}\, T$

INIT

$elems = [\![\,]\!]$

add

$\Delta(elems)$
$element? : T$

$elems' = elems \uplus [\![element?]\!]$

remove

$\Delta(elems)$
$element? : T$

$elems = elems' \uplus [\![element?]\!]$

contains

$element? : T$
$contains! : \mathbb{B}$

$contains! = element? \text{ in } elems$

Figure 5. Specification of the *Multiset* library class

Set[*T*]

Multiset[*T*]
$\restriction(add)$

$\forall x : \text{dom}\, elems \bullet count(elems, x) \leq 1$

add

$\Delta(elems)$
$element? : T$

$\neg\ (element? \text{ in } elems)$
$size' = size + 1$
$elems' = elems \uplus [\![element?]\!]$

Figure 6. Specification of the *Set* library class

parent and child library component classes. However the main difference is in the data representations. Whereas in the first example the query and library classes had the same data representation (after instantiation), in this example the data representations are fundamentally different (the query represented as a set, while the library component is represented as a multiset).

3.3. Matching the query against the library classes

The first step in matching the query class *Dictionary* against the library class *Set* is to identify a coupling invariant, which describes a relationship between attributes in the query and library classes. For this example we define the coupling invariant, *CI*, as:

$$CI(words, elems) \mathrel{\widehat{=}} words = \text{dom}\, elems$$

An adaptation, π, which gives a match between the query and library component is:

$$\pi = \{T \mapsto Word, add \mapsto insert, remove \mapsto delete, \\ element? \mapsto words?, contains! \mapsto indic!\}$$

As in the first example, the query class is matched against the library class by matching individual attributes and methods, with respect to the adaptation π, as well as with respect to the coupling invariant *CI*.

Attribute matching In contrast to the first example, where attributes were matched simply by renaming and instantiation and providing a one-to-one mapping between individual attributes, attribute matching for these advanced class matching techniques is more complex. Attribute matching involves proving that each state representable by the query Q can be represented by a state in the library specification class S. The following definition is based on a technique for matching state schemas in state-based modules [5, Definition 5.4].

Definition 3 *Given a query Q with attribute(s) $w : \tau$ and invariant Q_{inv}, and a library class S with attribute(s) $v : \gamma$ and invariant S_{inv}, then the attributes of Q and S are said to match with respect to a coupling invariant $CI(w, v)$ and adaptation π iff:*

$$\forall w : \tau \bullet Q_{inv} \Rightarrow \exists v : \gamma(\pi) \bullet CI(w, v) \land S_{inv}(\pi)$$

In the example, the matching condition, with respect to the coupling invariant *CI* and adaptation π from above, can be written as:

$$\forall words : \mathbb{F}\, Word \bullet \exists elems : \text{bag}\, Word \bullet \\ words = \text{dom}\, elems \land \\ \forall x : \text{dom}\, elems \bullet count(elems, x) \leq 1$$

Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 **IEEE**

This obligation can be easily satisfied; for a (finite) set $\{w_1, w_2, .., w_n\}$, we can represent this by the bag $[\![w_1, w_2, .., w_n]\!]$, which clearly satisfies the coupling invariant and the invariant of the class *Set*.

Matching the Init methods The method *Init* from the *Dictionary* query class is matched against the method *Init* from the inherited *Multiset* class. They match using the following version of *exact match*, extended to include matching with respect to an adaptation and a coupling invariant.

Definition 4 *A query class Q and a library component S are said to be an exact match, with respect to an adaptation π, and a coupling invariant CI, iff*

$$CI(Q_{in}, S_{in}) \Rightarrow (Q_{pre} \Leftrightarrow S_{pre}(\pi)) \wedge$$
$$CI(Q_{in}, S_{in}) \wedge CI(Q_{out}, S_{out}) \Rightarrow (S_{post}(\pi) \Leftrightarrow Q_{post})$$

where Q_{in} (S_{in}) refers to the initial (unprimed) state variables of Q (S), and Q_{out} (S_{out}) refers to the final (primed) state variables of Q (S).

The matching condition for the *Init* methods after the adaptation has been applied is as follows:

$$words = \mathrm{dom}\, elems \wedge words' = \mathrm{dom}\, elems' \Rightarrow$$
$$(elems' = [\![\,]\!] \Leftrightarrow words' = \varnothing)$$

Note that preconditions for initialisation methods are trivial, so have been omitted from the above matching condition. The proof of the remainder of the condition is straightforward.

Matching add and insert The *insert* method from the *Dictionary* class is matched against the *add* method from the *Set* class. Note that *add* is an inherited method, but is redefined in the child class *Set*. In matching queries against classes that redefine inherited methods, only the redefined method is used as a candidate during matching. So in this case we do not attempt to match *insert* against the *add* method from the *Multiset* class.

In matching *insert* against *add*, we use the following advanced version of *plug-in match with adaptation* (see Definition 2), which has been extended to include coupling invariants:

Definition 5 (Plug-in CI match with adaptations) *A query class Q and a library component S are said to be a plug-in match, with respect to an adaptation (σ,π), and a coupling invariant CI, iff*

$$CI(Q_{in}, S_{in}) \wedge Q_{pre} \Rightarrow S_{pre}(\pi) \wedge$$
$$CI(Q_{in}, S_{in}) \wedge CI(Q_{out}, S_{out}) \wedge S_{post}(\pi) \Rightarrow Q_{post}.$$

The two operations *add* and *insert* match under this definition, with respect to the adaptation (π) given above. The matching condition, after the adaptation has been applied throughout, becomes:

$$words = \mathrm{dom}\, elems \wedge word? \notin words \Rightarrow$$
$$\neg\, word? \,\mathrm{in}\, elems \wedge$$
$$words = \mathrm{dom}\, elems \wedge words' = \mathrm{dom}\, elems' \wedge$$
$$elems' = elems \uplus [\![word?]\!] \Rightarrow$$
$$words' = words \cup \{word?\}$$

The proof of the first conjunct in the matching condition is straightforward. To prove the remainder of the condition we observe:

$$\begin{aligned} words' &= \mathrm{dom}\, elems' \\ &= \mathrm{dom}(elems \uplus [\![word?]\!]) \\ &= \mathrm{dom}\, elems \cup \{word?\} \\ &= words \cup \{word?\} \end{aligned}$$

Matching remove and delete Suppose we attempt to match *delete* against *remove* using *plug-in CI match with adaptation*. The matching condition is:

$$words = \mathrm{dom}\, elems \wedge word \in words \Rightarrow \mathrm{true} \wedge$$
$$words = \mathrm{dom}\, elems \wedge words' = \mathrm{dom}\, elems' \wedge$$
$$elems = elems' \uplus [\![word?]\!] \Rightarrow$$
$$words' = words \setminus \{word\}$$

However this proof obligation is not provable; a counterexample is when *elems* contains more than one instance of the *word*. But in matching against a method from the *Set* library class (including inherited methods), we can assume that the state invariant holds before the call to *remove*, when matching postconditions. In this case the matching condition can be restated as:

$$words = \mathrm{dom}\, elems \wedge word \in words \Rightarrow \mathrm{true} \wedge$$
$$words = \mathrm{dom}\, elems \wedge words' = \mathrm{dom}\, elems' \wedge$$
$$\forall x : \mathrm{dom}\, elems \bullet count(elems, x) \leq 1 \wedge$$
$$elems = elems' \uplus [\![word?]\!] \Rightarrow$$
$$words' = words \setminus \{word\}$$

With these extra conditions, the counterexample can no longer occur, and the proof obligation can be discharged.

Matching contains and check The methods are matched using the extended version of exact match (Definition 4). The following proof obligation has to be satisfied.

$$words = \mathrm{dom}\, elems \wedge words' = \mathrm{dom}\, elems' \Rightarrow$$
$$indic! = word? \,\mathrm{in}\, elems \Leftrightarrow indic! = word? \in words$$

This can be proven by focusing on the left-hand side of the equivalence and then applying the coupling invariant,

Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 **IEEE**

i.e.:

$indic! = word? \text{ in } elems$
$\Leftrightarrow indic! = word? \in \text{dom } elems$
$\Leftrightarrow indic! = word? \in words$

4. Discussion

To date we have focussed on the theoretical aspects of matching object-oriented component specifications. In this section we will discuss some of the practical aspects related to implementing these specification matching techniques and using them for library retrieval. In general terms, retrieval relies on algorithms for matching individual units within queries and library components. Retrieval also relies on a decision logic that determines what order library components are matched, and what algorithms are used at each stage.

In implementing the matching algorithms we should be guided by current technologies used in functional specification matching. Broadly speaking we can split these algorithms into two groups that we will refer to as *syntactic based* matching and *semantic based* matching. Syntactic based algorithms [13, 6] are based on unification of expressions. Such algorithms are readily automated, and are well suited to calculating instantiations of parameters (including higher-order parameters). However these algorithms are limited by the fact that they can only handle structural equivalence of expressions (such as alpha-equivalence or associative commutative equivalence) and not more general forms of logical equivalence, or indeed the various forms of relaxed matching.

Semantic based matching methods [12, 7, 16] are based on matching specifications up to logical equivalence, or more general logical-based relaxed forms of matching. They rely on reasoning support, typically in the form of a theorem prover. The advantage of semantic based matching methods is the increase in precision and recall. In particular they offer support for equational reasoning. There are two main disadvantages of the current approaches to semantic based specification matching. Firstly the semantic based matching methods are based on first order logics, therefore these methods offer no support for higher-order parameters. Secondly semantic based reasoning relies more heavily on user interaction, therefore specification matching can become a major bottleneck in the retrieval process.

For the basic matching examples in this paper, which only involved first order parameters, retrieval tool support would be based on semantic based matching algorithms. In certain circumstances (e.g., for template classes that include higher order parameters), we anticipate that unification based algorithms will also be required. For each type of unit within a class there will be at least one corresponding matching algorithm. Since the language clearly differentiates between state schemas, initialisation schemas and operation schemas, it is easy to ensure that we only attempt to match units of the same kind. In some cases we may have multiple matching strategies for a particular type of unit, for example we may implement both exact matching and plug-in matching for operation schemas. In this case a decision needs to be made on what strategies to apply and in what order. Such decisions could be made automatically by implementing a decision logic within the retrieval tool. For such a decision to be made there must be some clear benefit in applying one strategy in preference to another. This will depend on the nature of the algorithm, for example if the implementation of exact matching is fully automatic we might choose it ahead of an interactive relaxed matching strategy.

Implementing tool support for the advanced matching strategies involving coupling invariants is more difficult. The main problem in this case is choosing a coupling invariant. This is a creative part of the process, and in general cannot be automated. However if we view retrieval as part of an interactive creative design process then this is not such a problem. Having defined a coupling invariant the remainder of the matching process can be handled using tool support as with basic matching.

5. Conclusions

In this paper we have investigated how specification matching techniques can be extended to handle object-oriented components. The main feature of object-orientation languages that introduces new issues to specification matching is inheritance. Single inheritance can be handled by restricting the units from the library class used in matching to those in the child class, together with those in the parent class that have not been redefined. Information hiding (that is using private and protected members of a class) can be incorporated into the specification matching framework in a similar manner by restricting the units from the library class used in matching. Where the language distinguishes between private and protected members, different matching techniques could be defined that either include or exclude protected members.

An issue that has not been addressed in this paper is multiple inheritance. Using a class that inherits from multiple classes raises several problems. One such problem is where the child class inherits a unit from two or more classes; in this case a decision has to me made as to which parent class to use. A similar choice might be made in deciding which of the units is used in the matching process. Alternatively, all units might be used in the matching process, but we only return those matches that match at most one of these units (similar to ONE-match as defined in [4]).

Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 IEEE

Acknowledgements

This work was funded by Australian Research Council Discovery Grant DP0208046, Compilation of Specifications.

References

[1] R.J.R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[2] C.J. Fidge. Contextual matching of software library components. In P. Strooper and P. Muenchaisri, editors, *Proceedings of Ninth Asia-Pacific Software Engineering Conference (APSEC'2002)*, pages 307–316. IEEE Computer Society Press, December 2002.

[3] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison-Wesley, 1983.

[4] D. Hemer and P. Lindsay. Specification-based retrieval strategies for module reuse. In D. Grant and L. Stirling, editors, *Proc. of Australian Software Engineering Conference (ASWEC'2001)*, pages 235–243. IEEE Computer Society, August 2001.

[5] David Hemer. Specification matching of state-based components. Technical Report 02-34, Software Verification Research Centre, The University of Queensland, Australia, 2002.

[6] David Hemer and Peter Lindsay. Supporting component-based reuse in CARE. In Michael Oudshoorn, editor, *Proceedings of the Twenty-Fifth Australasian Computer Science Conference*, volume 4 of *Conferences in Research and Practice in Information Technology*, pages 95–104. Australian Computer Society Inc., January 2002.

[7] J-J. Jeng and B.H.C Cheng. Specification matching for software reuse: A foundation. In *Proceedings of ACM Symposium on Software Reuse*, pages 97–105, April 1995.

[8] K. Melhorn and S. Naher. Leda: A platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, January 1995.

[9] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[10] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[11] John Penix and Perry Alexander. Toward automated component adaptation. In *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering*, pages 535–542, June 1997.

[12] D.E. Perry and S.S. Popovich. Inquire: Predicate-based use and reuse. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pages 144–151, September 1993.

[13] E.J. Rollins and J.M. Wing. Specifications as search keys for software libraries. In K. Furukawa, editor, *Eighth International Conference on Logic Programming*, pages 173–187. MIT Press, June 1991.

[14] H.W. Schmidt and W. Zimmermann. A complexity calculus for object-oriented programs. *Journal of Object-Oriented Systems*, 1(2):117–147, 1994.

[15] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods Series. Kluwer Academic Publishers, 2000.

[16] A. Moormann Zaremski and J.M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering*, 6(4):333–369, October 1997.

Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 **IEEE**