On Merging Object-Oriented Formal Specifications

FATHI TAIBI[1]
FOUAD MOHAMMED ABBOU[2]
MD.JAHANGIR ALAM[2]

[1]University of Tun Abdul Razak, Selangor, Malaysia
[2] Multimedia University, Selangor, Malaysia
[1]`taibi@unitar.edu.my`
[2]`(fouad,md.jahangir.alam)@mmu.edu.my`

Abstract. Collaborative development allows the delegation of tasks among developers, which eases the development of complex software systems. The software artifacts created because of this collaboration need to be merged at the end of a particular development activity. To achieve this, a merging approach must be able to produce results that are correct, complete and consistent. Furthermore, the merge approach must rely on a strong similarity detection technique that allows determining the similarities that exist between the different artifacts. Merging requirement specifications allows discovering and dealing with inconsistencies at an early stage which reduces both the time and effort associated with such task compared to dealing with them at later stages such as during design or deployment. This paper proposes an approach for merging Object-Oriented (OO) formal specification views of a given system. The proposed approach is redundancy-aware, and uses a heuristic matching approach to find the correspondences between the views. Finally, the approach's performance is empirically evaluated.

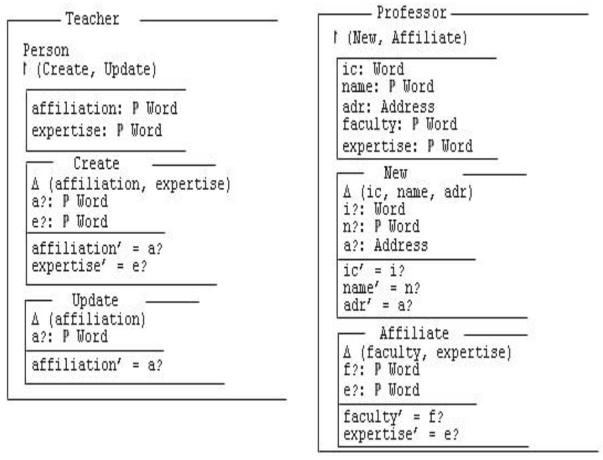
Keywords: collaboration, merging, formal methods, object-oriented.

(Received July 15, 2008 / Accepted November 05, 2008)

1 Introduction

The complexity of today's software systems makes collaborative development necessary to accomplish tasks as it allows the delegation of work among developers, which eases the development of such systems. This complexity is associated with the fact that organizations often attempt to construct large, complex software systems because of the availability of powerful processors at low prices. Frameworks are needed to allow developers perform their tasks independently and collaboratively. The major concern in such frameworks is the merging [6] [2] operation that allows combining the partial software views (such as software models or source code) created during the collaborative work. Furthermore, certain activities such as requirements specification require the participation of several people with different perspectives while the requirements themselves are derived through several stakeholders with different views, which increase the need to support the merging of such models. Merging requires the determination of the similarities that exist between the different views before combining them. The accuracy of the latter operation is crucial to producing a correct and consistent merge model. Missed matches lead to redundancies in the merged model, and the latter is one of the basic forms of inconsistency [3]. Furthermore, during merging, other conflicts between the views need to be discovered and resolved to produce a consistent result. Merging software requirements allows discovering and dealing with inconsistencies at an early stage which reduces both the time and effort associated with such task compared to dealing with them at later stages such as during design or deployment.

Merging requirements specified informally is tremendously difficult and error prone due to the ambiguous and misleading nature of natural languages and the notations used. Formal methods [1] offer a better alternative because of their precise and accurate nature, which makes it possible for automatic verification through model checking [4]. OO formal specifications have a double advantage as they combine the strengths of formal and OO methods. Thus, reuse is possible because of the OO nature of the developed specifications. The formal specification language Object-Z [10] is an OO extension of the well-established formal specification language Z [11], which makes it a good candidate to be used during the collaborative development of software specifications. As an example, Figure1 shows Object-Z classes taken from two specification views of a component of a university management system.



```
Teacher
Person
↾ (Create, Update)

affiliation: ℙ Word
expertise: ℙ Word

Create
Δ (affiliation, expertise)
a?: ℙ Word
e?: ℙ Word
affiliation′ = a?
expertise′ = e?

Update
Δ (affiliation)
a?: ℙ Word
affiliation′ = a?
```

```
Professor
↾ (New, Affiliate)

ic: Word
name: ℙ Word
adr: Address
faculty: ℙ Word
expertise: ℙ Word

New
Δ (ic, name, adr)
i?: Word
n?: ℙ Word
a?: Address
ic′ = i?
name′ = n?
adr′ = a?

Affiliate
Δ (faculty, expertise)
f?: ℙ Word
e?: ℙ Word
faculty′ = f?
expertise′ = e?
```

Figure 1: Two Views of an Object-Z class

The class *Teacher* is derived from another class *Person* that specifies the common features of a person (whether he is a student or a teacher). In the class *Teacher*, only the operations *Create* and *Update* are visible outside the class. The operation *Create* is used to assign values to the state attributes *affiliation* and *expertise*. While the operation *Update* is used to change the *affiliation* of a teacher in case he (she) has shifted to a new department.

The class *Professor* includes two operations *New* and *Affiliate* that are the only elements visible outside the class. The operation *New* is used to assign values to the state attributes *ic*, *name* and *adr*, which represents a professor's personal details. The operation *Affiliate* is used to assign values to the state attributes *faculty* and *expertise*. This operation can also be used to change the latter attributes in case they have already been assigned.

Clearly, the classes *Teacher* and *Professor* are quite similar. However, measuring their similarity in a precise manner requires the usage of an approach that makes use of the information available in the classes themselves as well as the classes related to them in their respective views. Identifying the similarities between the classes and elements of specification views is a prerequisite for successful merging. Figure 2 shows a preview of the specification views from which the classes *Teacher* and *Professor* are taken.

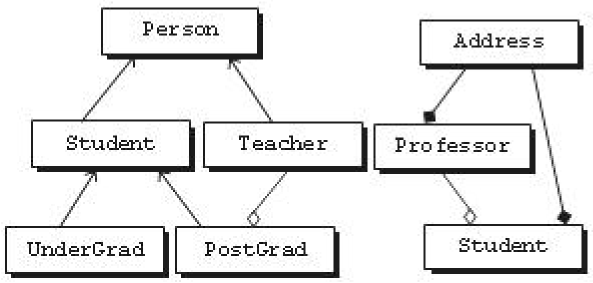


Figure 2: A Preview of Two Specifications of the Same System

Apart from the classes *Teacher* and *Professor*, the classes *Student* in both views represent a match and must be merged. In addition, the class *Person* might have a certain degree of similarity with the classes *Professor* and *Student* of the second view. Furthermore, several attributes, operations and relationships in the specification views could be found similar which requires merging them as well.

This paper proposes an approach for merging OO formal specifications represented according to a proposed meta-model. The proposed merging algorithm is redundancy-aware, and uses a heuristic matching approach to find the correspondences between the specifications. The next section presents a meta-model for representing specifications. This is followed by discussing a new matching approach that enables the determination of similarities between specifications. After that, a merging approach is proposed to combine these specifications. The approaches are empirically evaluated and related work is discussed. The final section concludes the paper and discusses future work.

2 Representing the Specifications

Before discussing the approaches that allow matching and merging OO formal specifications, there is a need to define how these specifications are represented. The proposed framework considers each specification as a graph $G = (V, E)$ where:

- V is a set of vertices (i.e. classes)

- E is a set of edges (i.e. classes' relationships) where each edge $e = (s, t, p)$:
  - s is the source vertex of e
  - t is the target vertex of e
  - $p =$ {inherited_by, aggregated_by, associated_with} is the type of the edge e.

Furthermore, each note in the graph G is associated with a graph $C = (N, A)$ where:

- N is a set of vertices (i.e. class' elements)
- A is a set of edges (i.e. relationships between elements) where each edge $a = (s, t, p)$:
  - s is the source vertex of a
  - t is the target vertex of a
  - $p =$ {is, used_by} is the type of the edge a.

Each element in the graphs G and C is mapped to an element in a proposed meta-model (M) that considers a specification as a set of interconnected vertices of type *Class*. The interconnection between these vertices represent the major relationships between the classes namely inheritance, aggregation/composition and association. Each vertex of type *Class* is associated with a graph whose vertices and edges represent class' elements and their relationships respectively. Figure 3 shows a proposed meta-model for OO formal specifications.

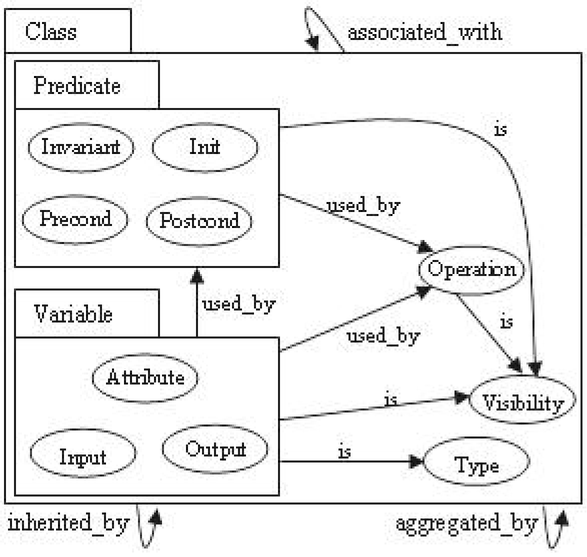


Figure 3: A Meta-Model for OO Formal Specifications

A class is modeled as a set of *Variable(s)* manipulated by *Operation(s)* containing some *Predicate(s)*. The proposed meta-model uses the generic semantics of formal specifications to differentiate between the different predicates, i.e. *Invariant*, *Init*, *Precond*, and *Postcond*. Furthermore, it uses the same semantics to differentiate between the different type of variables such as class' *attribute*, operation's *Input/Output*, and the *Data Type* of a variable is also taken into account. Finally, the *Visibility* of a class' element is used to highlight the public members of a class. Figure 4 shows the graphs representing the specification view of the class *Teacher* as well as the class itself.

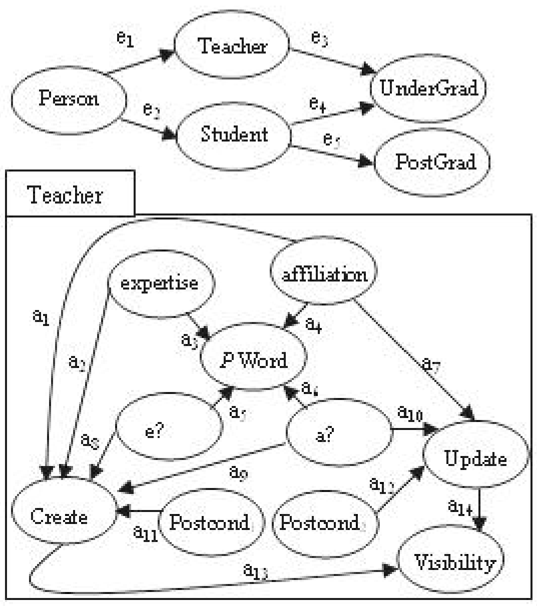


Figure 4: Graph Representation of a Specification View

In Figure 4, the edges e_1 and e_2 indicate that the class *Person* is *inherited_by* the classes *Teacher* and *Student* while the edges a_1 and a_7 indicate that the attributes *affiliation* is accessed (*used_by*) by the operations *Create* and *Update*. In addition, the edges a_{13} and a_{14} indicate that only the latter operations are visible outside the class *Teacher*. Furthermore, each node is typed according to its association with the meta-model, which allows processing them differently in the proposed matching and merging approaches.

3 Matching specifications' Elements

A merge algorithm requires the determination of the similarities that exist between the elements of the input specifications. These similarities are computed based on the syntactic (name) similarities that exist between the specifications' elements as well as their structure (content) similarity. The syntactic similarity between two elements is used to identify the early correspondences between the specifications and cannot alone be the basis of any matching approach.

The structural similarity is computed based on the structure of the elements to be mapped. The structure in this context does not only include the content of these elements but also the similarity of their ancestors, descendents as well as their neighbors. For example, in case of classes, the base classes could be treated as ancestors, the child classes could be treated as descendents, and classes involved in similar relationships could be treated as neighbors.

Computing an accurate similarity value between two specification elements requires the usage of several approaches because no particular approach is better than others are. Thus, we propose the usage of three additional concepts for which experiment results have shown an improvement in accuracy and efficiency. The full details of these experiments are discussed in [13]. The first concept is the use of early landmarks. Where for a particular specification element, a domain expert (or a user) could set the element(s) of the second specification that should be compared to it. This helps in improving the approach's efficiency as it reduces the number of comparison. It also helps reducing the probability of many false matches, which leads to a better precision. These landmarks could also be used to represent confirmed matches. The creation of landmarks is enabled by the automatic generation of high-level representations of the specification views (i.e. the classes and their relationships) as well as supporting browsing capabilities to explore their content. The second concept is automatically indexing the classes of the specifications as they are created. An index could be the most frequent attribute(s) used in a class, and if used in the computation of structural similarity, it could improve the accuracy of the matching results.

The final concept is the use *Mutual Enforcing Relationship* concept, i.e. after computing the overall similarity between classes, the results is used back to recompute the similarity of the classes' elements. In other word, *classes are similar if they have similar elements and elements are similar if they are contained in similar classes*. This could improve the accuracy of the matching results of class elements such as attributes and operations.

The proposed matching approach progresses in a *bottom-up* style as class' elements are compared before top-level elements. It also switches to *top-down* when the overall similarity of classes is known to re-compute the similarities of their elements (*Mutual Enforcing Relationship*).

Formally, the matching relation *Match* is defined as: $Match : ELEMENT \times ELEMENT \times TYPE \rightarrow \mathbb{R}$ Where for two elements (e_1, e_2) of type *t* such as *Class*, *Variable*, or *Operation*, $Match(e_1, e_2, t)$ returns a real number between 0 and 1 representing the overall similarity between them. This overall similarity must be bigger than a chosen threshold T (a real number between 0 and 1) that represents the strictness of the match relation. The overall similarity is a normalized value of the structural similarity of (e_1, e_2) by their syntactic similarity. It is computed using the following formula: $(S_{Syntactic} + S_{Structural})/(1 + S_{Syntactic})$

Given two strings X and Y, the syntactic similarity $S_{Syntactic}$ between them is obtained by taking the maximum value from the Longest Common Substring (LCS) and 2-gram algorithms respectively [12]. For both algorithms, the similarity metric is defined as: $2 * Length_{Same}/Length_{All}$

$Length_{Same}$ is the cardinality of the longest common substring between X and Y for LCS. It is the cardinality of the set containing similar substrings of size-2 obtained from X and Y for *2-gram*. $Length_{All}$ is the cardinality of the set comprising the disjoint union of the characters of both strings or the disjoint union of the substrings of size-2 obtained from both strings for LCS and *2-gram* respectively.

Given two specification elements, $S_{Structural}$ between them is calculated using the following formula: $2 * sum/(sum + count)$

Where sum is obtained by cumulating the syntactic similarities (or 0 or 1) between all the compatible items of the elements and $count$ is the number of items used in the calculation of sum. Table 1 illustrates how to compute the overall similarity between the operation *Create* of the class *Teacher* and the operations *New* and *Affiliate* of the class *Professor*.

Table 1: Similarity Computation Example

	Create vs. New	Create vs. Affiliate
Visibility	1	1
Classes	0.125	0.125
Accessed Attributes	0.686	0.735
Inputs	0.686	0.95
Outputs	1	1
Preconditions	1	1
Postconditions	0.666	1
$S_{Syntactic}$	0.222	0.4
$S_{Structural}$	0.737	0.907
Overall Similarity	0.785	0.934

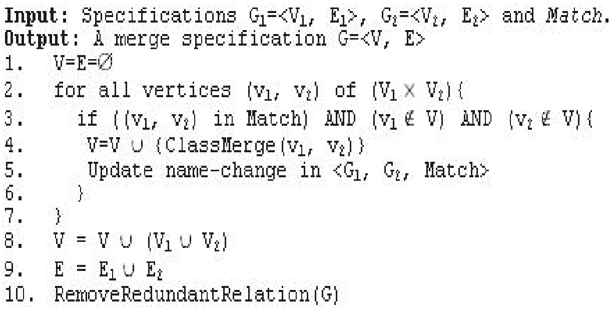
During the computation of $S_{Structural}$, attribute names (as well as inputs and outputs) are replayed by their respective type for predicates (Inits, invariants, preconditions, and postconditions). The reason behind this normalization is that for all the latter elements; type is the most important factor; names as well as their order of appearance could be ignored in this context. Thus, the impact of $S_{Syntactic}$ on $S_{Structural}$ is reduced for behaviorally similar classes and operations. For example, the postconditions of the operations *Create* and *Affiliate* are totally (1) similar in this context as they manipulate the same type of data and perform exactly the same actions.

If two operations A and B are matched, their parameters (inputs/outputs) are matched according to the similarities of the attributes they manipulate. For example, the operations Create and Affiliate represent a match assuming a threshold $T = 0.8$. Thus, the parameter $a?$ is matched to $f?$ as they manipulate matched attributes (*affiliation* and *faculty* in this case). The same intuition is applied to the argument $e?$ of the two operations. Finally, it is important to note that arguments with same names and same types do not represent a match unless they manipulate matched attributes.

4 A Proposed Merging Approach

The proposed merging algorithm takes as input two OO formal specifications (represented by graphs G_1 and G_2) and creates a merge specification (graph G) combining their content based on their matching relation (*Match*). The merge specification must be correct, complete, and consistent. Correctness is achieved by ensuring that the merge specification preserves the properties of all specification views by combining the right elements. Completeness is achieved by including all (matched and non-matched) elements contained in the specification views. Finally, consistency is ensured by identifying and resolving the conflicts between the specification views. Redundancy is one of the basic forms of inconsistency that should be dealt with first.

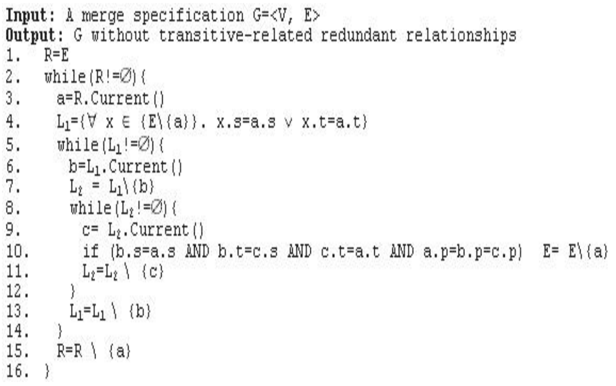
The proposed algorithm uses the information contained in Match to combine first all the matched elements of G_1 and G_2 and then their non-matched elements are added to G. Figure 5 shows the proposed merging algorithm.



```
Input: Specifications G1=<V1, E1>, G2=<V2, E2> and Match.
Output: A merge specification G=<V, E>
1.  V=E=∅
2.  for all vertices (v1, v2) of (V1 × V2){
3.     if ((v1, v2) in Match) AND (v1 ∉ V) AND (v2 ∉ V){
4.      V=V ∪ {ClassMerge(v1, v2)}
5.      Update name-change in <G1, G2, Match>
6.     }
7.  }
8.  V = V ∪ (V1 ∪ V2)
9.  E = E1 ∪ E2
10. RemoveRedundantRelation(G)
```

Figure 5: SpecificationMerge Algorithm

The algorithm starts by combining all the matched vertices (classes) of G_1 and G_2 and adding them to G (line 4) by calling another algorithm *ClassMerge*. The latter algorithm is discussed below. Each time two classes (v_1, v_2) are merged, the name used for the merge class is used to update (align) the specifications (G_1 and G_2) as well as their matching relation (*Match*). The name of the first class could be used as a merge name in case the order of the input specifications reflects their priority. After that, all the unmatched classes of G_1 and G_2 are added to G (line 8). All edges of G_1 and G_2 are added to G (line 9). The algorithm ensures that merged specification is free from redundant vertices or edges before applying the unions in lines 8 and 9 as the input specifications are aligned each time two classes are merged. The first inconsistency taken into account is the redundancy of class' relationships based on their transitive nature (algorithm called in line 10). Figure 6 shows the algorithm that removes this kind of redundancies.



```
Input: A merge specification G=<V, E>
Output: G without transitive-related redundant relationships
1.  R=E
2.  while(R!=∅){
3.    a=R.Current()
4.    L1={∀ x ∈ (E\{a}). x.s=a.s ∨ x.t=a.t}
5.    while(L1!=∅){
6.      b=L1.Current()
7.      L2 = L1\{b}
8.      while(L2!=∅){
9.        c= L2.Current()
10.       if (b.s=a.s AND b.t=c.s AND c.t=a.t AND a.p=b.p=c.p)  E= E\{a}
11.       L2=L2 \ {c}
12.     }
13.     L1=L1 \ {b}
14.   }
15.   R=R \ {a}
16. }
```

Figure 6: RemoveRedundantRelation Algorithm

The algorithm accepts as input a graph G with a set of edges E and removes from it all redundant edges originated because of transitivity. Inheritance, composition and aggregation relationships are all transitive. Thus, if an edge $'a'$ links a class X with a class Z, and there exists two other edges $'b'$ and $'c'$ of the same type sharing the same source/target class as $'a'$ linking the classes X with a class Y and Y with Z respectively, the edge $'a'$ is removed from E (line 10). This process is repeated for all the elements of E, and at the end of it, E is free from this kind of redundancies.

```
Input: Classes C1=<N1, A1>, C2=<N2, A2> and their matching relation Match.
Output: A merge class C=<N, A>
1.  N=A=∅
2.  for all the elements (n1,n2) of (N1 × N2) {
3.    if ((n1,n2)in Match) OR (compatible(n1,n2)) AND (n1∉ N) AND (n2∉ N){
4.      N=N ∪ {ElementMerge(n1,n2)}
5.      Update name-change in <G1, G2, Match>
6.    }
7.  }
8.  N=N ∪ (N1 ∪ N2)
9.  A = A1 ∪ A2
```

Figure 7: ClassMerge Algorithm

Merging matched classes is done by combining their matched/compatible elements and including the remaining elements in the merge class. Figure 7 shows the algorithm used to merge two matched classes.

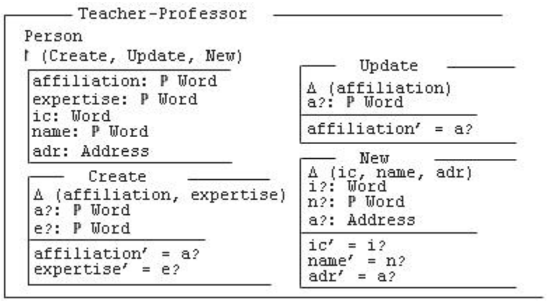
The algorithm accepts two matched classes C_1 and C_2 and returns their merge class C. It starts by merging their matched (such as attributes and operations) and compatible (such as invariants) elements by calling *ElementMerge* algorithm (line 4). Each time two elements are merged, the name change is updated in the specifications G_1 and G_2 as well as their match relation Match (line 5). After that, the non-matched elements are added to the merge class (line 8). Finally, edges of C_1 and C_2 are added to C (line 9) while the algorithm guarantees non-publication of edges and nodes because of specification alignment. *ElementMerge* accepts two class elements (n_1, n_2) and combines them according to their respective type.

Figure 8 shows a recursive definition of *ElementMerge*. Merging two class elements (n_1, n_2) is performed according to their type. For variables with compatible (or same) types (line 4), the type of the merge element n will be the super-type of the types of (n_1, n_2) (line 6). In case the types of (n_1, n_2) are not compatible, *ElementMerge* simply returns both elements (line 8). For operations, their matched (such as inputs/outputs) and compatible elements (such as pre and post conditions) are merged using a recursive call to *ElementMerge* (line 19). The pre-condition of the merge operation will be the disjunction of pre-conditions of the respective operations (line 13). The post-condition of the merge operation as well as other predicates such as invariants is the conjunction of their items (line 14). Finally, the remaining non-matched elements are simply added to the merge operation (line 21).

The merging approach has been tested with several

```
Input: Two elements n1 and n2.
Output: A merge element n
1. n=∅
2. switch(Typeof(n1, n2)) {
3.   case "Variables" {
4.    if Compatible(Type(n1), Type(n2)) {
5.     Name(n) = n1
6.     Type(n)=SuperType(Type(n1),Type(n2))
7.    }
8.    else n={n1, n2}
9.   }
10.     case "Predicates"{
11.      P={ ∀ xi. n1=(x1 ∧ x2 ∧ … ∧ xn) }
12.      Q={ ∀ yj. n2=(y1 ∧ y2 ∧ … ∧ ym) }
13.      if Precond(n1,n2)     n = n1 ∨ n2
14.      else { n =(z1 ∧ z2 ∧ … ∧ zk). ∀ zk ∈ P ∪ Q }
15.     }
16.     case "Operations" {
17.      for all items (i,j) of (n1 × n2) {
18.       if ((i,j)in Match) OR Compatible(i,j)) AND (i∉ n1) AND (j∉ n2)
19.         n= n ∪ ElementMerge(i,j)
20.      }
21.      n= n ∪ (n1 ∪ n2)
22.     }
23.   }
```

Figure 8: ElementMerge Algorithm



```
Teacher-Professor
Person
↾ (Create, Update, New)
  affiliation: ℙ Word
  expertise: ℙ Word
  ic: Word
  name: ℙ Word
  adr: Address

  Create
  Δ (affiliation, expertise)
  a?: ℙ Word
  e?: ℙ Word
  ----------
  affiliation′ = a?
  expertise′ = e?

  Update
  Δ (affiliation)
  a?: ℙ Word
  ----------
  affiliation′ = a?

  New
  Δ (ic, name, adr)
  i?: Word
  n?: ℙ Word
  a?: Address
  ----------
  ic′ = i?
  name′ = n?
  adr′ = a?
```

Figure 9: A Example of a Merged Class

specifications and Figure 9 shows the class obtained by merging the classes *Teacher* and *Professor* introduced earlier. The class *Teacher-Professor* incorporates all the matched and non-matched elements of the merged classes. The operation *Create* represents the result of merging of the operations *Create* and *Affiliate* of the merged classes. The operations *Update* and *New* of the merged classes are not matched, thus directly included. The attributes *affiliation* and *expertise* represent the merging of (affiliation, faculty) and (expertise, expertise) respectively. Finally, the attributes *ic*, *name* and *adr* are directly included as they are not matched. It is important to note that since the class *Teacher* is derived from a class *Person*, the merge class *Teacher-Professor* is consequently derived from the same class. This may lead to redundancies as the attributes *ic*, *name*, and *adr* might be defined in the class *Person* (which is the case in this example). This is also valid for the operation

New. Dealing with this kind of redundancies is out of the scope of this paper. An independent approach need to be proposed to deal with this kind of inconsistency as well as with other forms of inconsistencies such as ensuring that the asymmetric and acyclic properties of class' relationships are maintained.

5 Evaluation

A merge approach is useful if it produces accurate (correct and complete) results with cheap processing means (i.e. time and space) while ensuring a consistent result. The complexity of the proposed merging approach is around $O(n * m)$ where n and m represent the number of elements in the two specifications views respectively. The matching approach has similar complexity that could be further reduced if landmarks between the specifications views are created. The complexity becomes $O((n - k) * (m - k))$ where k is the number of landmarks created. The matching and merging approaches are effective if they do not produce too many false matches and do not miss too many correct matches. *Precision* and *recall* metrics were used in the evaluation. *Precision* measures quality and is the ratio of correct matches/merges found to the total number of matches/merges found. *Recall* measures coverage and is the ratio of the correct matches/merges found to the total number of all correct matches/merges. The proposed approaches have been intensively evaluated based on several small/medium sized case studies. One of them (Figure 2) includes two specifications with 62 elements and 92 relationships. Precision and recall were computed for a threshold ranging from 0.5 up to 0.9. Figure 10 shows the results obtained.

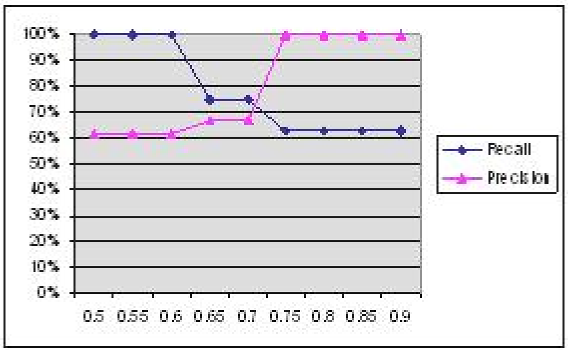


Figure 10: Results For the University Management System

For low to medium threshold (0.5 - 0.7), the matching approach has shown good recall (75% - 100%) combined with an acceptable precision (62% - 67%). For high threshold (0.75 - 0.9), the approach has shown a perfect precision (100%) combined with an acceptable recall (63%). It is important that the matching approach is capable to identify as many correct matches as possible because it is easier to remove incorrect matches compared to identifying missing ones manually. Thus, the results obtained were good because for a reference threshold of 0.71 (where the precision and the recall graphs intersect) both the precision and recall scored 72%. The latter figure indicates that only 28% of similar elements have not been matched and that 72% of the matched elements are indeed correct. The missed matches need to be identified by a domain expert along with the removal of the incorrect ones leading to a merge result that is correct and complete. The amount of evaluation done so far serves only as a proof-of-concept. More (intensive) evaluation is needed to tune and validate the proposed approach.

6 Related Work

In [7], merging hierarchical statechart models based on a correspondence relation is done by processing shared and non-shared elements separately. The merge model contains the shared elements of the input models as normal behaviors and the non-shared elements as variabilities that are represented using parameterization. The latter are guarded by conditions denoting the origin model before added to the merge model. To ensure that the merge model is behaviorally sound and deterministic, the proposed approach requires the shared states of the input models to have identical events, conditions, and priorities. This leads to unnecessary redundant transitions, and the approach did not include an automatic way to deal with them, i.e. identify and handle them. In addition, the proposed merge operator is based on a heuristic match operator whose average precision was relatively low (around 50%). Thus, a domain expert needs to manually identify and remove half of (incorrect) matches, which lowers the benefits of any implementation of the merge approach.

In [14], a differencing algorithm is proposed to detect the structural changes between the designs of subsequent versions of OO software. The input models are obtained through reverse engineering of two corresponding code versions. The algorithm reports the differences between them in terms of additions / removals, moves, and renaming of program elements such as packages, classes, interfaces, fields, and methods. The differencing algorithm computes an overall similarity based on name and structure similarity metrics. The name similarity metric was computed using 2-gram algorithm. The latter does not provide good results in case of short strings as well as in case of long strings where a substring has been replaced by a different word. This could affect the accuracy of the early landmarks based on which structural and overall similarities are computed. In addition, the proposed algorithm assumes that enough design entities remain the "same" between the two consecutive versions of the system. The latter assumption is weak in the sense that there is no guarantee that the developers of the new version of the system do not violate it.

In [5], a generic algorithm was proposed to differentiate between UML models. The first phase of the approach consists of identifying correspondences (i.e. similarity) while the second phase deduces the appropriate differences between the documents based on their correspondences. The output produced consists of a table with structural differences, attribute differences, reference differences, and move differences between the input documents. The calculation of similarity between two elements starts by comparing the sub-elements first (bottom-up). The similarity metric used assigns different weights to the elements of the input document with the highest weight given to the similarity of class names (0.4). This weight is inappropriate as it does not permit matching classes with poor name similarity but with very similar content (attributes and operations). While the LCS used to compare names does provide accurate results in case of a change of word order, which affects the recall of the proposed algorithm.

Finally, in [9] and [8], a consistency checking approach (from a structural perspective) is proposed for homogeneous models. The approach is intended to deal with consistency rules beyond pairwise checking. The approach consists of creating a merged model and then checking consistency rules written using Relational Manipulation Language (RML). The merge operator used considers each model as a graph and each mapping as a binary relation over two models equating their corresponding elements. The approach lacks efficiency because there is a need to store external links to equivalence groups mapping elements of the input models. Furthermore, the approach applicability is limited to classes (for class diagrams) and entities (for entity relationship diagrams) and their relationships only. It does not indicate how class entities/elements are supposed to be merged. Finally, the proposed framework indicates that the correspondences between the input models are explicitly specified. This may work for small models but not for a big number of real-size models. The correspondences between the input models should be identified automatically (or semi-automatically).

7 Conclusion

An approach to merge OO formal specifications was proposed in this paper. It uses the results of a matching approach that is responsible of identifying the similarities between the elements of the input specifications. The specifications are represented using a proposed meta-model that is based on the generic semantics of both formal and OO methods, which makes the proposed merging approach useful for a wide range of applications. The matching approach uses heuristics for both syntactic and structural similarities with the possibility of adjusting the results before merging starts. The latter adjustment consists of removing incorrect matches and identifying missed ones. High recall combined with high precision is a pre-requisite to make the adjustments minimal. The proposed matching approach has shown good precision and good recall for reasonably high threshold, which provides a good basis for the merging approach. The proposed merging approach comprises algorithms for combining specification views (classes, relationships and class-elements) and is redundant-aware for class relationships as well as for class elements.

The scalability of the proposed approaches needs to be further tested as all the experiments have included small/medium sized case studies. Thus, specifications views for large and complex systems are needed to study this scalability.

Finally, identifying and resolving all the inconsistencies in the merge specification is the first issue to be dealt with after more validation work is put on the merging approach.

References

[1] Bowen, J. and Hinchey, M. Ten commandments revisited. a ten-year perspective on the industrial application of formal methods. In *10th ACM IWFMICS conference*, pages 8–16, 2005.

[2] Fortsch, S. and Westfechtel, B. Differencing and merging of software diagrams - state of the art and challenges. In *ICSDT 2007 conference,*, pages 90–99, 2007.

[3] Gervasi, V. and Zowghi, D. Reasoning about inconsistencies in natural language requirements. *ACM TOSEM*, 14(3):277–330, 2005.

[4] Kassel, G. and Smith, G. Model checking object-z classes: Some experiments with fdr. In *APSEC 2001 conference*, pages 445–452, 2001.

[5] Kelter, U., Wehren, J., and Niere, J. A generic difference algorithm for uml models. In *Software Engineering Conference*, pages 105–116, 2005.

[6] Mens, T. A state-of-the-art survey on software merging. *EEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[7] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., and Zave, P. Matching and merging of statecharts specifications. In *ICSE'2007*, pages 54–64, 2007.

[8] Sabetzadeh, M. and Easterbrook, S. View merging in the presence of incompleteness and inconsistency. *Requirements Engineering Journal*, 11(3):174–193, 2006.

[9] Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., and Chechik, M. Consistency checking of conceptual models via model merging. In *RE'2007*, pages 21–230, 2007.

[10] Smith, G. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[11] Spivey, J. *The Z Notation*. Prentice Hall, 2nd edition, 1992.

[12] Taibi, F., Abbou, F., and Alam, M. A matching approach for object-oriented formal specifications. *Journal of Object Technology*, 7(8):139–153, 2008.

[13] Taibi, F., Abbou, F., and Alam, M. Towards identifying similarities between formal specification views. In *ICIMU'08*, pages 789–794, 2008.

[14] Xing, Z. and Stroulia, E. Differencing logical uml models. *Journal of Automated Software Engineering*, 14(2):215–259, 2007.