JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2008**

Vol. 7, No. 8, November-December 2008

A Matching Approach for Object-Oriented Formal Specifications

Fathi Taibi, University of Tun Abdul Razak, Selangor, Malaysia
Fouad Mohammed Abbou, Multimedia University, Selangor, Malaysia
Md Jahangir Alam, Multimedia University, Selangor, Malaysia

Abstract

Software merging is needed at different stages of software development to combine the artifacts created or modified by the parallel work of the different developers involved in the project. An accurate matching approach is the key to successful software merging as well as to conflicts identification. In this paper, a new matching approach for Object-Oriented formal specifications is proposed. Object-Z is used as a specification language. However, the proposed approach is meant to be applicable to a wide range of Object-Oriented software artifacts. Merging formal requirements specifications is motivated by the fact that it could help in identifying (and resolving) conflicts that will cost higher to identify (and resolve) at later stages of software development. The proposed approach incorporates heuristics for both syntactic and structural similarity. The empirical results obtained through a prototype implementation of the proposed approach were very encouraging.

1 INTRODUCTION

Software artifacts incorporate the views of several developers working on a software project, which makes software merging essential for successful large-scale software development. Merging [Mens02] is needed at different development phases to combine the partial artifacts created or modified by the parallel work of the different developers involved in the project. For example, during integration, source code merging happens at regular intervals to combine the work carried out by different programmers. The same thing could be said about requirements specifications and design models. Merging software artifacts manually is tremendously difficult, time consuming, error prone, and very expensive. Thus, it must be automated.

In addition to promoting collaboration and distributed development, when used at an early stage of the software development such as during requirements specification, the merging process helps in identifying and resolving conflicts that will cost higher to identify and resolve during later stages of development such as during testing or integration.

Fathi Taibi, Fouad Mohammed Abbou, Md Jahangir Alam: "A Matching Approach for Object-Oriented Formal Specifications", in *Journal of Object Technology*, vol. 7, no. 8, November-December 2008, pp. 139-153 http://www.jot.fm/issues/issue_2008_11/article3/

Merging requirements specified informally (by textual or graphical means) is difficult and time consuming because of the ambiguous nature of natural languages and the notations used. Moreover, informal specifications are sometimes misleading. Formal methods offer a better alternative because of their precise and accurate nature. One of these methods is Object-Z [Smith00] that combines the strengths of two worlds: the world of formal languages and the world of Object-Oriented (OO) methods. When used to specify systems' requirements, Object-Z produces specifications that are clear, precise, and object-oriented.

Matching [Nejat07] the elements of the software artifacts to be combined is required before merging. Matching is based around the concept of similarity. An accurate matching approach is the foundation of any successful merging mechanism. Matching allows identifying the correspondences between the artifacts, which helps in discovering the inconsistencies [Nusei01] between them. This makes it possible to deal with those conflicts so that the output from the merge process is consistent. In addition, other than software merging and consistency checking, an accurate matching approach has several other applications such as information retrieval, software reuse and evolution, etc.

In the following sections, first we introduce formal specification using Object-Z. This is followed by discussing matching software artifacts by highlighting the means needed to achieve it and the issues related to those means. After that, we proposed a new matching approach for Object-Oriented formal specifications. The approach is then empirically evaluated. This is followed by discussing related work, and the last section concludes the paper and discusses future work.

2 FORMAL SPECIFICATION USING OBJECT-Z

Object-Z is an OO extension of the well-established formal specification language Z [Spive92]. It is a state-based formal specification language in which system states, initial states, and operations are modeled by schemas comprising a set of variable declarations constrained by predicates. Figure 1 shows the components of an Object-Z class.

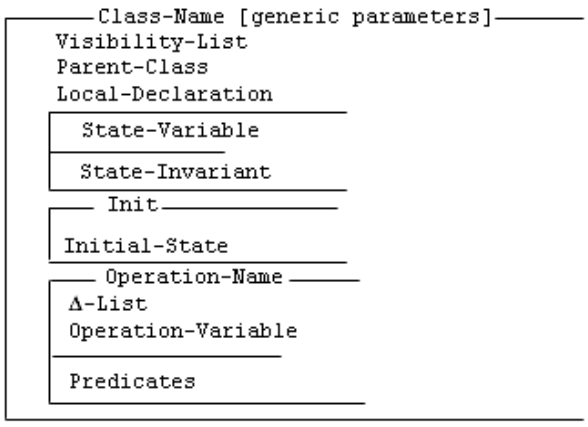


Figure 1: An Object-Z class

The Visibility-List states the class members that are visible outside the class (same as public in Java). If the visibility list is omitted then all the class' features are visible. Visibility lists are not inherited, i.e. the derived class may nominate any inherited feature in its own visibility list. Parent-Class is a list of all the parent (base) classes. Local-Declarations contains the class attributes, and constants declarations with their restrictions in some cases. An invariant is a property that must be satisfied by the class objects all the time. The value of a class attribute must be compatible with the class invariant and the invariants of all its parent classes. The state schema comprises two parts, State-variable declarations, and a predicate section (State-Invariant) defining the invariant of the class. State-Invariant places restrictions on State-Variable, and may make use of the Local-Declarations as well. The Init schema is used to specify the initial values of the state variables. An operation schema comprises four sections. Operation-name is a String representing the operation's name. Operation-variable contains the declaration of the inputs and the outputs of the operation. Θ-List contains a list of the state variables that will be changed by the operation, i.e. the state variables that are not listed in the Θ-List are unchanged. The predicate section of an operation contains a pre-condition that must be satisfied in order to execute a post-condition.

The following figure shows an example of two different views of an Object-Z class.

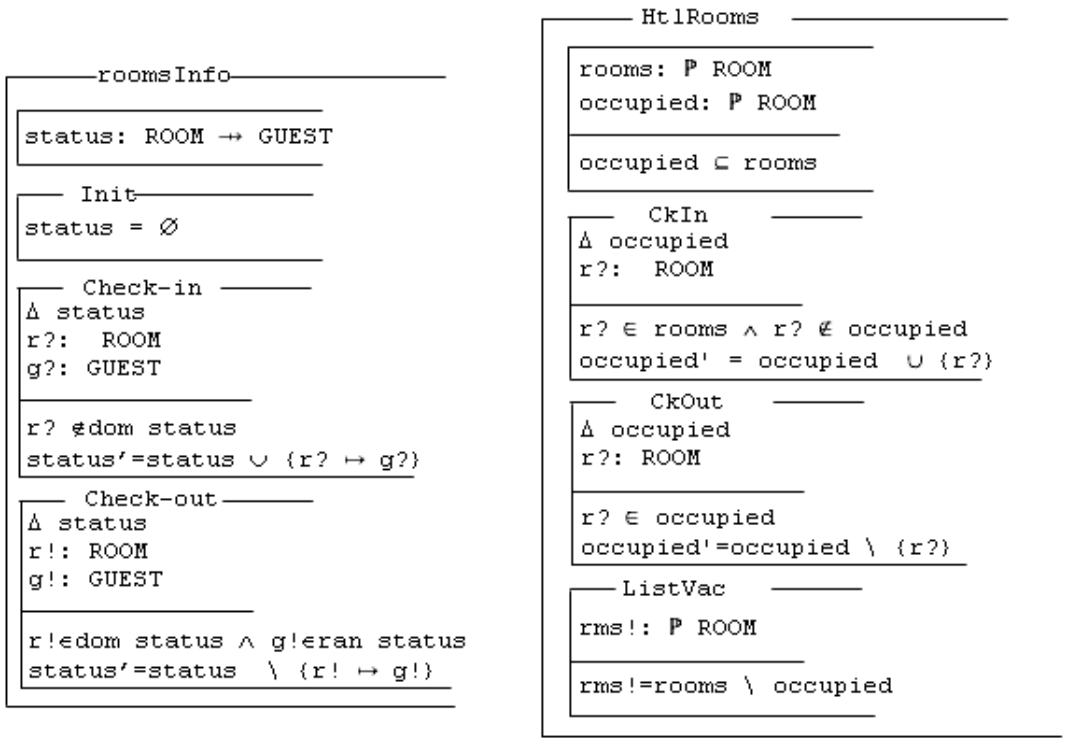


Figure 2: Two views of an Object-Z class

The classes roomsInfo and HtlRooms represent two views of the same class taken from two specifications of a simple hotel management system. In the class roomsInfo (figure 2a), all the elements are visible outside the class as there is no visibility list. status is a state variable declared as a partial function from ROOM to GUEST. Initially, all the rooms are vacant (status=∅). The operation Check-in and Check-out are used to add or remove elements from status thus changing it value (Δ status). Checking in a guest g? into a room r? requires verifying that the room is not occupied (r? ∉ dom status). A guest g! occupying a room r! (r! ∈ dom status ∧ g! ∈ ran status) checks out at the end of his stay at the hotel, and his association with the room is then removed from status. The symbols '?' and '!' are used to highlight inputs and outputs respectively while '' is used to highlight the new value of an attribute.

An accurate matching approach should identify *roomsInfo* and *HtlRooms* as a positive match. However, relying on name similarity between them (or their elements) will not provide an accurate answer. The following sections discuss the matching process and how to compute all kind of similarity metrics between two OO formal specifications based on a new proposed approach.

3 MATCHING

Matching is based around the concept of similarity. Syntactic (name) similarity could be verified by comparing strings, such as class names, operation names, attributes, signatures, etc. The Longest Common Substring (LCS) algorithm [Gusfi99] could be used to compute a similarity metric between two strings. Its overall performance is acceptable. However, it does not give accurate results in case of a change of word order, which often happens when dealing with software artifacts created by different developers. For example, a developer could name an attribute emptyrooms while another one could use roomsempty for the same purpose. In this case, the two attributes should be identified as a match. However, based on the LCS algorithm, their similarity is only 0.5.

The *N-gram* algorithm [Manni99] caters for this kind of limitations as it considers the number of identical substrings of length N. Hence, the similarity between *emptyrooms* and *roomsempty* based on the *2-gram* algorithm is 0.8. However, intensive random experiments have shown that the *2-gram* algorithm does not provide accurate results in case of short strings. For example, *rm* and *rms* are 0.66 similar based on 2-gram while they are 0.8 similar based on *LCS*. In addition, the 2-gram algorithm does not provide good matching results in case of long strings where a substring has been replaced by a different word. For example, the similarity between *emptyrooms* and *roomsunoccupied* is 0.27 while it is 0.4 based on *LCS*.

For a matching approach to provide accurate results, it cannot rely solely on syntactic similarity to find correspondences between the artifacts to be merged. For example, the syntactic similarity between the classes *roomsInfo* and *HtlRooms* of figure 2 is 0.588 and 0.533 according to the *LCS* and *2-gram* algorithms respectively. Syntactic similarity should be merely used as starting point for the matching approach. Taking the

maximum returned value from *LCS* and *2-gram* will be sufficient to start a structural similarity algorithm. The initial syntactic matches will form the early landmarks based on which the structural matching process operates.

Structural similarity incorporates several aspects of OO artifacts. It could be computed by checking the ancestors, the descendants, the relationships, the structure (elements), and the parameters (signatures) of the items to be matched. In the case of formal specifications, predicates such as invariants, preconditions, and postconditions could be used in the process. More elements used in the matching process will likely have a positive effect on the accuracy of the matching process. Thus, combining syntactic with structural similarity could help precisely indicating whether there is a match or not.

4 A MATCHING APPROACH FOR OBJECT-ORIENTED FORMAL SPECIFICATIONS

The approach

Given two specifications S_1 and S_2, the matching approach computes an overall similarity metric between all the classes of S_1 and those of S_2. The matches are identified based on a chosen threshold. Figure 3 shows the matching algorithm.

```
Match
Input: 2 specifications S1=< A1, A2,…, An>, S2=< B1, B2,…, Bm>, and a
threshold T
Output: A correspondence relation R storing the positive matches
between S1 and S2
1.   R=∅
2.   for all classes Ai of S1
3.    for all the classes Bj of S2
4.     x= overallSimilarity(Ai, Bj)
5.     if x>=T then
6.      R=R ∪ {(Ai, Bj)↦ x}
7.     else (Remove all elements from R associated with (Ai, Bj))
8.     end if
9.    end for
10.  end for
11.  return R
```

Figure 3: The matching algorithm

Match makes a call to an overall similarity algorithm *(overallSimilarity)* that return a value between 0 and 1 that combines both syntactic and structural similarity. The call is made between all the classes of the two specifications (line 4), which results in an *O(nm)* complexity. The classes whose overall similarity is bigger or equal to a chosen threshold (line 5) are added to the correspondence relation (line 6). The latter will be an input to the merging process. The higher the threshold is, the stricter the similarity requirement is. After a match is identified and added to the correspondence relation R, the class B_j is not

discarded from the next round of comparison. Non-removal of a class B_j with a confirmed match in S_1 is motivated by the fact that it is up to the merging process (or a domain expert) to normalize the correspondence relation by choosing the best possible match among the correspondences available in case of multiple matches for a given class.

In addition to the matched classes, the correspondence relation R stores the correspondences between the elements of the matched classes during the computation of the overall similarity. In case the overall similarity between two classes is below the threshold, all the elements added to R during the computation of the overall similarity (i.e. associated with the current classes (A_i, B_j)) are removed (line 7). Typically, an OO artifact is also a set of inter-related classes, which makes proposed matching approach useful for wide range of OO artifacts.

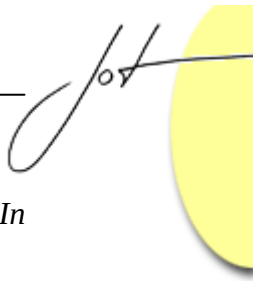
The similarity algorithms

The proposed matching approach combines the *LCS* and *2-gram* algorithms to compute a syntactic similarity metric between the to-be-matched elements or their items. Given two strings X and Y, the algorithm returns a value between 0 and 1 corresponding to the syntactic similarity between them. The computation is case-insensitive and all the non-relevant characters such as space are not taken into account. The following figure shows the proposed algorithm.

```
synSimilarity
Input: 2 String X and Y
Output: A value between 0 and 1
1.   X=X.toUpper()
2.   Y=Y.toUpper()
3.   X=X.removeAllSpace()
4.   Y=Y.removeAllSpace()
5.   L1=X.getAllPairs()
6.   L2=Y.getAllPairs()
7.   same= L1 ∩ L2
8.   all= L1 ∪ L2
9.   a=2*same.length()/all.length()
10.  b=2*LCS(X, Y)/(X.length()+Y.length())
11.  return Max(a, b)
```

Figure 4: Syntactic similarity algorithm

As discussed previously, both the *LCS* and *2-gram* algorithms have their own limitations. However, the experimental results obtained have shown that when the LCS algorithm provides a low similarity value for a known positive match then the 2-gram algorithm provides a better result and vise versa. Thus, we compute two similarity values (lines 9 and 10) based on the 2-gram and LCS algorithms respectively and return the maximum value (line 11) among them. *LCS(X, Y)* returns the length of the longest common substring between the parameters X and Y. For the 2-gram algorithm, we first need to generate a list of substrings of size 2 for all the adjacent characters of each string (lines 5 and 6). Then we compute their intersection and disjoint union (lines 7 and 8) that are used



to compute the intended metric. For example, *synSimilarity* between *Check-in* and *CkIn* returns 0.4 (i.e. *Max(0.4, 0.33)*).

The structural similarity is computed based on the structural sameness between two classes (or two class' elements such as operations). The following table summarizes all the aspects that are taken into account.

Table 1: The elements used to compute structural similarity

Item	Description
Class_name	The name of the class
Visibility_list	The public attributes and operations of the class
Inheritance_list	The ancestor(s) of the class
Relationship_list	The aggregated/composed class(es)
Attributes	Local and state declarations
Invariant	Local declaration predicate combined with state invariant.
INIT	The predicate of the INIT schema.
Operation_list	The operations of the class
Inputs / Outputs	The inputs/outputs of an operation
pre/post conditions	The pre/post conditions of an operation

The above list corresponds to the elements of an Object-Z class based on which a structural similarity is computed. Most of the elements used have equivalences in other OO artifacts. For example, the INIT schema corresponds to a constructor of a class, a list of operations corresponds to a list a methods of a class in a program or a class diagram, the inputs / outputs of an operation correspond to the arguments of a method, etc. The predicates (invariants, pre-conditions, and post-conditions), which are not available in other OO artifacts such as class diagrams; provide a positive added advantage to the matching process for both classes and operations.

When computing structural similarity, we propose the use of the following rules:

- A class name contributes to the structure similarity of all its elements when compared with the elements of other classes.
- For attributes and invariants, each attribute name is replaced by its respective type.
- For inputs/outputs and pre-conditions/post-conditions, each argument (and attribute) name is replaced by its respective type.
- When an element is missing from one (or both) class (es) of interest, the default value taken is the syntactic similarity between the classes (i.e. their root).

The above rules are motivated by the fact that when comparing two set of attributes or arguments, the type is the most important factor. Names as well as their order of appearance could be ignored. In addition, for invariants, pre-conditions and post-conditions we apply the same technique as what matters when comparing two sets of predicates is to see how similar they are regardless to the identifiers used. Finally, the class name, should contribute to the structural similarity of all its elements.

The following figure shows the complete structural similarity algorithm divided into two sections, one for classes and second for class' elements such as attributes, operations, visibility lists, etc.

```
structuralSimilarity
Input: 2 classes C1, C2
Output: a value (between 0 and 1) and a correspondence relation R
1.   S1= C1.getElements()
2.   S2= C2.getElements()
3.   sum=0
4.   count=0
5.   If S1.size()=0 OR S2.size()=0 return synSimilarity(C1.Name, C2.Name)
6.   for all elements Ei of S1
7.    for all elements Fj of S2
8.     if compatible(Ei, Fj) then
9.      a= strSimilarity(Ei, Fj)
10.     R= R ∪ (( Ei, Fj) ↦ a)
11.     sum=sum+a
12.     count++
13.     S2.removeElement(Fj)
14.    end if
15.   end for
16.  end for
17.  return (2*sum)/(sum+count)
 /*-------------------------------------------------------------------------*/
strSimilarity
Input: 2 elements E1 and E2
Output: a value (between 0 and 1)
1.   L1= E1.getItems()
2.   L2= E2.getItems()
3.   s=0
4.   c=0
5.   If L1.size()=0 OR L2.size()=0 return synSimilarity(E1.root, E2.root)
6.   for all items Ik of L1
7.    for all items Jm of L2
8.     if compatible(Ik, Jm)then
9.      s+=synSimilarity(Ik, Jm)
10.     c++
11.     L2.removeElement(Jm)
12.    end if
13.   end for
14.  end for
15.  return (2*s)/(s+c)
```

Figure 5: Structural similarity algorithm

The first part of the algorithm describes how structural similarity is computed for two classes. If one of the classes (or both) are empty, then the algorithm returns their syntactic similarity (line 5). Otherwise, it processes all the elements of the first class by comparing them with their compatible equivalent elements of the second class using strSimilarity. In each round, the computed metric is added along with the elements to the correspondence relation R (line 9), and it is accumulated into sum (line 11) that will be used (along with the number of correspondences count) to return the result at the end of the process (line

17). Once a class' element is processed, it is discarded from the next round of comparisons (line 13) to speed up the process.

The second part of the algorithm describes how to compute a structural similarity between the elements of two classes. In case one of the elements (or both) is not available, the algorithm returns the syntactic similarity between the names of the two classes (line 5). For example, when one of the classes does not have a visibility list, the algorithm returns the syntactic similarity of their classes, etc. The algorithm checks the correspondences between the items of the elements to be matched and computes a normalized value (`((2*s)/(s+c))`) using *synSimilarity* (accumulated into s) as well as the number of correspondences (c). Once a correspondence between two elements is established, the element is discarded from the next round of comparisons (line 11) to increase the efficiency of the algorithm.

The overall similarity between two given classes is calculated using a normalized value based on both syntactic and structural similarity. Figure 6 shows the overall similarity algorithm.

```
overallSimilarity
Input: 2 classes C1, C2
Output: a value (between 0 and 1)
1. m1=synSimilarity(C1, C2)
2. m2=structuralSimilarity(C1, C2)
3. return (m1+ m2)/(m1+1)
```

Figure 6: Overall similarity algorithm

The normalized value (m1+ m2)/(m1+1) is always between 0 and 1. It reflects a higher similarity metric compared to the average. For example if m1=0 (i.e. there is no syntactic similarity between the two classes) and m2=0.7, the above algorithm return 0.7. Finally, the chosen threshold will decide whether two classes represent a positive match or not. The following section analyzes the empirical results obtained through a prototype implementation of the approach.

Empirical evaluation

The objective from evaluating the proposed approach is to find out its usefulness for developers facing a matching problem. In case of small models, developers may find it easy to identify the matches manually. The proposed approach is intended to provide a quick and accurate way to identify matches when matching by hand is not possible (or hard to achieve). This is true for a wide range of OO specifications intended for medium or large-scale software. Moreover, even for small specifications with complex internal relationships between its components, matching by hand is a tremendous task.

The result from the match process will be used for merging as well as for consistency checking of the OO formal specifications of interest. If done manually, this is always going to be tremendously difficult. Hence, the proposed matching approach is useful if it produces accurate results with cheap processing means (i.e. time & space).

The complexity of the proposed matching approach is *O(mn)* where m and n represent the number of classes in the two specifications. This complexity could be reduced if we discard the matched class of the second specification (or the matched classes of both specifications) from the next round of comparison. However, we chose to leave both matched classes, as it is possible to find better matches during the next rounds. In addition, we propose to normalize (automatically or by a domain expert) the correspondence relation *R* before the start of the merging process.

The matching approach is effective if it does not produce too many incorrect matches (false positives) and does not produce too many missed matches (false negatives). We employ *precision* and *recall* metrics in our evaluation. *Precision* is the ratio of correct matches found to the total number of matches found. It measures *quality*. *Recall*, which measures *coverage* (low number of false negatives), is the ratio of the correct matches found to the total number of all correct matches. A good matching technique should produce high precision and high recall. However, it is generally agreed in this field that the latter is hard to achieve but it is possible to obtain good balanced results.

We have evaluated our approach based on several small/medium size case studies. One of them includes the classes of figure 2, and includes four other classes namely *staffInfo* and *transactionInfo* for the first view and *HtlTransactions* for the second view. The following table summarizes some characteristics of the two views studied for the hotel management system.

Table 2: Characteristics of the studied specifications

	#Classes	**#Operations**	**#Relationships**	**Total number of matches**
View 1	3	6	2	7
View 2	2	5	1	

We computed the precision and recall for a threshold ranging from 0.5 up to 0.9. Figure 7 shows the results obtained using *overallSimilarity*.

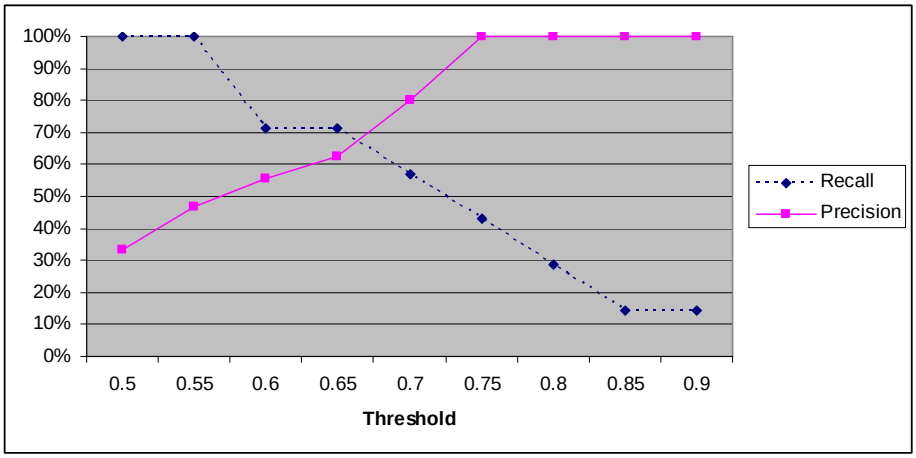


Figure 7: Overall matching results for the hotel management system

The results obtained confirmed our initial assumptions that elements with strong syntactic similarity are more likely to be confirmed as matches. In addition, the elements with low syntactic similarity such as *roomsInfo* and *HtlRooms* (0.588) where confirmed as match through structural similarity (for *roomsInfo* and *HtlRooms*, it was 0.72). The overall similarity between *roomsInfo* and *HtlRooms* was 0.82. Low threshold (0.5 up to 0.55) has resulted in perfect recall (100%) and low precision (33%-47%). Medium threshold (0.6 up to 0.7) has resulted in acceptable recall (57%-71%) and acceptable precision (56%-80%). Finally, high threshold (0.75 up to 0.9) has resulted in low recall (14%-43%) and perfect precision (100%).

A threshold of 0.7 (that is not too low or too high) has given the best-balanced results as four out of the five matches confirmed where correct (precision is 80%) knowing that the model contains seven matches (recall is 57%). With the same threshold, we have obtained results that were consistent with the latter using different (but similar-sized) case studies.

It is important to compare the results obtained with the overall similarity to those obtained using syntactic similarity only. The following figure highlights the results obtained.

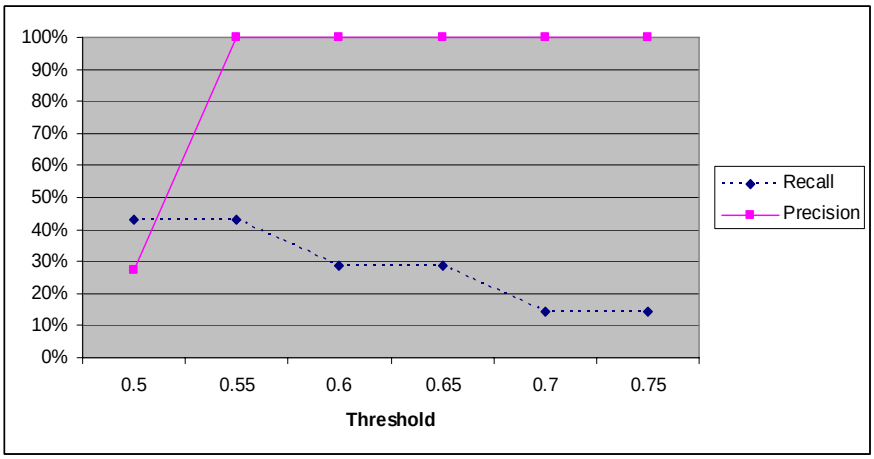


Figure 8: Syntactic matching results for the hotel management system

Recall was very low (43% down to 14%), which is in agreement with our initial hypotheses as syntactic similarity can only be used as a starting point for the matching process and cannot identify all the possible matches. Recall was 0% for high threshold (0.8 up to 0.9). On the other hand, for low threshold (0.5), precision was low (27%) as only three matches where correct out the eleven identified. For threshold between 0.55 up to 0.75, syntactic matching showed a perfect (100%) precision. Precision could not be computed for high threshold (0.8 up to 0.9) as no matches were identified.

The overall performance of the proposed approach was good from a precision and efficiency perspectives. In addition, the number of false negatives was acceptable, which provides a good platform for the merging and consistency checking processes.

5 RELATED WORK

In [Xing05], an algorithm is proposed to automatically detect structural changes between the designs of subsequent versions of an object-oriented software. It uses class diagrams obtained by reverse-engineering from a java software system. It reports the differences between the models in terms of additions, removals, changes, and renamings. In [Apiwa04] and [Forts07], similar differencing methods were proposed for object-oriented programs and software diagrams respectively.

In [Nejat07], two operators (match and merge) for model management were proposed. The proposed operators manipulate hierarchical Statecharts. The proposed match operator makes use of static and behavioral properties. It proposes the use of sanity checks for the match results obtained before applying the merge operator. Thus, it argued that the merging process should be made semi-automated. Indeed, it might be reasonable to make the matching process interactive where user seeds are used to confirm the most obvious relations as well as to rectify incorrect relations. This will likely to improve the precision of the approach.

In [Sabet07], model merging was used to check the consistency of conceptual models. The proposed approach constructs a merge model then verifies it against some consistency constraints of interest. The consistency diagnostics obtained over the merge are projected back to the original models and their relationships. The approach focused on conceptual modeling formalisms with graphical notations, and the consistency checking rules were described using the Relational Manipulation Language (RML).

In [Kelte05], a generic algorithm was for differencing between UML models was proposed. The proposed algorithm computes differences between UML models encoded as XMI files. The approach is generic in the sense it covers a broad range of UML diagrams. The results obtained on small documents showed good runtimes. The approach was tested mainly on class diagrams and Statecharts. The proposed approach suffers from the lack of accuracy of XMI specifications. Finally, [Mens02] has provided an intensive, critical, and throughout review of software merging. It provided recommendations on merging approaches and tools. They include precision, efficiency, easy implementation, conflicts detection, and the use of three-way merging as it provides better results compared to two-way merging. However, it is not always possible to use three-way merging, such as during the specification of the requirements of a new system. Here developers create different views of the requirements but no common ancestor between them is available.

Our proposed approach is inspired from both [Xing05] and [Nejat07]. However, we chose to focus on requirements specification rather than design artifacts or source code because it allows the detection of conflicts at an early stage, which improves the quality of the developed software. We employ formal OO specifications as they help capturing precisely (and in an OO style) the software requirements. We combined the strength of both LCS and 2-gram algorithms to compute syntactic similarity. The latter is combined with a structural similarity metric computed using a generic algorithm to produce an overall similarity metric that is used in matching process. Finally, we have included all

the generic aspects of OO software artifacts to make the proposed approach useful for a wide range of applications.

6 CONCLUSION AND FUTURE WORK

In this paper, a new generic approach for matching the elements of OO formal specifications was proposed. The approach is intended to be used for software merging and consistency checking. Merging formal OO specifications was motivated by the fact that it would help identifying conflicts at an early stage of development and promote collaborative development. The approach combines syntactic and structural similarity algorithm to perform matching. The results obtained through intensive testing of a prototype implementation of the approach were encouraging as they incorporated good precision (for reasonable threshold) combined with cheap processing time (complexity).

Object-Z was used to specify the different views. However, the proposed approach could be used to match any other OO software artifacts due to the generic properties based on which the similarity algorithms are founded. Furthermore, the matching results between the specifications of interest are stored in a correspondence relation that could be reviewed and adjusted by a domain expert before the start of the merging and consistency checking processes.

Because the proposed approach was mainly tested using small/medium sized specifications, the next step would be to study its performance for industry-sized specifications with hundreds of classes and elements. Finally, it would be interesting also, to use the proposed approach for matching elements between OO programs.

REFERENCES

[Apiwa04] T. Apiwattanapong et al, *A Differencing Algorithm for Object-Oriented Programs*, Proceedings of the 19th International Conference on Automated Software Engineering, pp. 2-13, 2004.

[Forts07] S. Fortsch et al., Differencing and Merging of Software Diagrams: State of the Art and Challenges, Second International Conference on Software and Data Technologies, INSTICC, 2007.

[Gusfi99] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1999.

[Kelte05] U. Kelter et al., A Generic Difference Algorithm for UML Models; In Proceedings of the SE 2005, 2005.

[Manni99] C. Manning et al., Foundations of Statistical Natural Language Processing, MIT Press, 1999.

[Mehra05] A. Mehra et al., A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design, In Proceedings of the 2005 ACM/IEEE International Conference on Automated Software Engineering, pp 204-213, 2005.

[Mens02] T. Mens, A State-of-the-Art Survey on Software Merging, IEEE Transactions on Software Engineering, Vol. 28, No. 5, pp. 449-462, 2002.

[Nejat07] S. Nejati et al., Matching and Merging of Statecharts Specifications, 29th International Conference on Software Engineering (ICSE'07), 2007.

[Noron07] A. Boronat et al., Formal Model Merging Applied to Class Diagram Integration, Electronic Notes in Theoretical Computer Science, Vol. 166, pp. 5–26, 2007.

[Nusei01] B. Nuseibeh et al., Making Consistency Respectable in Software Development. Journal of Systems and Software, Vol. 58, pp. 171-180, 2001.

[Potti03] R. Pottinger et al., Merging Models Based on Given Correspondences, In Proceedings of the 29th international conference on VLBD, 2003.

[Sabet07] M. Sabetzadeh et al., Consistency Checking of Conceptual Models via Model Merging. 15th IEEE International Requirements Engineering Conference (RE'07), 2007.

[Smith00] G. Smith, The Object-Z Specification Language, Kluwer Academic Publishers, 2000.

[Spive92] J. Spivey, The Z notation – A Reference Manual, Prentice Hall, 2nd Edition, 1992

[Xing05] Z. Xing et al, UMLDiff: An Algorithm for Object-Oriented Design Differencing, In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp. 54-65, 2005.

[Zarem97] A. Zaremski et al., Specification Matching of Software Components, ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 4, pp. 333-369, 1997.

About the authors

Fathi Taibi is a lecturer at the Faculty of Information Technology of the University of Tun Abdul Razak. His research interests include formal specification, Object-Oriented methods, distributed development, and software verification. He can be reached at taibi@unitar.edu.my.

Dr. Fouad Mohammed Abbou is an associate professor (Alcatel-Lucent) attached to the Faculty of Engineering of Multimedia University. His research interests include optical systems, optical networks, and quantum communications. He can be reached at fouad@mmu.edu.my.

Dr. Md Jahangir Alam is a lecturer at the Faculty of Information Technology of Multimedia University. His research interests include image processing, pattern recognition, and artificial intelligence. He can be reached at md.jahangir.alam@mmu.edu.my.