JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2010**

Vol. 9, No. 1, January-February 2010

On Differencing Object-Oriented Formal Specifications

Fathi Taibi, University of Tun Abdul Razak, Malaysia
Md. Jahangir Alam, Multimedia University, Malaysia
Junaidi Abdullah, Multimedia University, Malaysia

Abstract

Requirements specification is a collaborative activity that involves several developers specifying the requirements elicited through several stakeholders. Operation-base merging allows combining specifications using the information available about their state as well as their evolution or change. Thus, leading to a more precise, accurate and efficient merging. Differencing specifications is a tedious, complicated, and a crucial process needed for operation-based merging of specifications resulting from collaboration. An approach for differencing Object-Oriented formal specifications is proposed in this paper. The difference is modeled as a set of primitive operations and is produced based on the matching results of specifications' elements. These matchings are calculated based on an approach employing elements' syntactic and structural similarities. The proposed differencing approach is empirically validated.

1 INTRODUCTION

Collaboration [17] is necessary during the development of large-scale software systems where several developers work in parallel on different aspects of the same system. Often, this leads to the creation of different but related documents. These documents could be in the form of design models, software specifications, source code, etc. During a particular collaborative activity, a resulting local version of a document needs to be merged [2, 6] with the version of the document available in a shared repository. The latter shared document encloses all the modifications made locally by the developers involved and checked into the repository at that point in time.

Specifying software requirements is an important, complicated and error prone task that involves the collaboration of several peopled specifying requirements elicited through several stakeholders. Studies have shown that most of the problems with software projects such as not meeting the needs of stakeholders, late delivery and budget overrun can be traced back to problems with the requirements [11]. Asynchronous collaboration allows members of a group to modify copies of a shared specification in

Fathi Taibi, Fouad Mohammed Abbou, and Md. Jahangir Alam: "On Differencing Object-Oriented Formal Specifcations", in *Journal of Object Technology*, vol. 9, no. 1, January - February 2010, pp. 183 - 198, http://www.jot.fm/issues/issue_2010_01/article5/

isolation, working in parallel and afterwards synchronizing their copies to reestablish a common view. This gives a great deal of flexibility, and matches the needs of collaborative requirements specification. Moreover, since requirements specification is an early phase of software development, any conflicts detected and resolved at this phase will have a positve impact on the quality and cost of the delivered software as these conflicts will cost higher to detect and resolve during later stages of development.

Merging requirements specified informally is unpractical, inefficient, error prone, and time consuming due to the ambiguous and imprecise nature of natural languages and most of the graphical notations used. Formal methods [8] offer a better alternative because of their precise and accurate nature. Object-Oriented (OO) formal methods, such as Object-Z [16], combine the strengths of two worlds: the world of formal languages and the world of OO methods. When used to specify software requirements, they produce specifications that are precise, clear, and highly reusable. Thus, they are suitable to be used when developing specifications collaboratively as they facilitate systematic manipulation.

Employing unique identifiers for the elements of the merged documents; such as in [13]; introduces tool dependency. The latter term refers to approaches that are dependent on the tools used to create and modify the documents to be merged. In order to support the tool independence requirement [5], merging should not rely on elements' unique identifiers. Thus, there is a need for an approach that can compute the exact difference between two versions of a document.

In operation-based merging [9, 12], the difference (or delta) is modeled as a set of primitive operations transforming a document's version into another one. Analyzing these deltas provides a good support for the systematic detection and resolution of conflicts [14] before merging their content with a shared document. Moreover, operation-based merging leads to a better efficient as the number of operations differencing documents is statistically smaller than the number of elements they contain.

Several existing differencing approaches; such as in [3, 4, 10]; process the manipulated documents as trees, which is restrictive and not applicable to a large number of documents including software requirements specifications. Moreover, even if the similarity detection approaches used work well with trees, there is no guarantee that they will be able to detect the similarities between the elements of documents with a graph structure such as OO formal specifications. Furthermore, in [12], it has been indicated that domain independent approaches do not to work well with all software documents compared to approaches intended for specific documents. Finally, to our knowledge, there is no existing differencing approach intended specifically for OO formal specifications.

A differencing approach for OO formal specifications is proposed in this paper. It comprises two parts: the first part consists of comparing specifications to identify their matching elements, and the second part uses the matching results to produce deltas differentiating them. These deltas are formally modeled as a set of primitive operation with traceability information allowing the reversal of their effect. The differencing approach is empirically validated.

2 MATCHING THE ELEMENTS OF SPECIFICATIONS

Differencing specifications requires identifying their matching elements. As a motivation example, consider the following classes representing a shared specification, and two versions representing some parallel modifications made to it by two different developers. Object-Z notation has been used to specify the three versions.

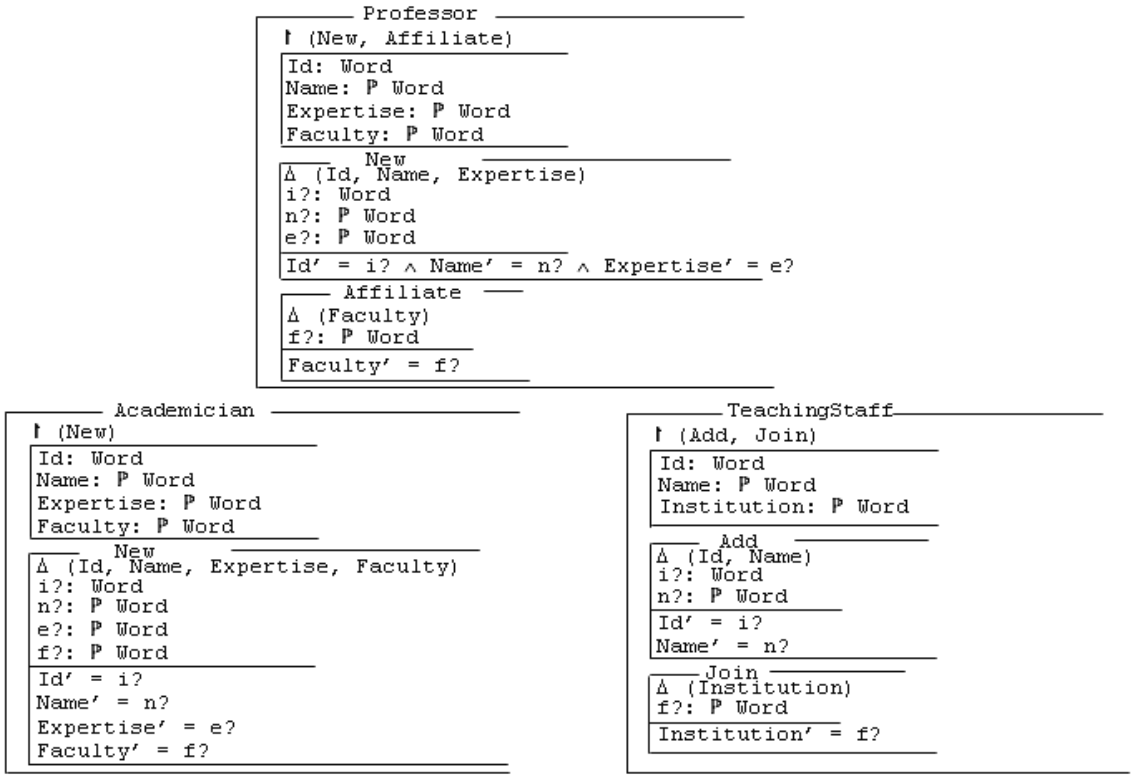


Figure 1: Three versions of an Object-Z class

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 *Id*, *Name* and *Expertise*, which represents a professor's personal data. The operation *Affiliate* is used to assign a value to the state attribute *Faculty*. The classes *Academician* and *TeachingStaff* are the result of some parallel modifications made to the class *Professor* by two different developers. In the class *Academician*, in addition to the class name that has been changed, the operation *Affiliate* has been removed and its functionality (dealing with the attribute *Faculty*) has been delegated to the operation *New* that is the only class' element visible. In the class *TeachingStaff*, in addition to the class name that has been modified, the attribute *Expertise* has been removed, the attribute *Faculty* has been renamed as *Institution* while the operations *New* and *Affiliate* have been renamed as *Add* and *Join* respectively. In addition, the operation *New* has been modified by removing the part dealing with the deleted attribute *Expertise*.

A good matching approach should be able to produce precise and accurate results identifying the similarities between the elements of the compared specifications. For example, it should match the classes *Professor* and *TeachingStaff*, the attributes *Faculty* and *Institution*, the operations *New* and *Add*, the operations *Join* and *Affiliate*, etc.

In the proposed matcing approach, each input specification is treated as a graph whose nodes are the specification's elements. Each link has a source and a target element as well as a type. For example in case of an operation *O* defined in a class *A*, a link is created to represent this relation. Its source and target are *O* and *A* respectively, and its type is "declared_in". This is further discussed in the next section.

The similarities that exist between specifications' elements are stored in a matching function:

$$\text{Match: ELEMENT} \times \text{ELEMENT} \rightarrow \mathbb{R}$$

ELEMENT is the base class representing specifications' elements that include *Class*, *Variable*, *Operation*, and *Predicate*. The returned value of *Match* is a real number (between 0 and 1) representing the exact similarity that exist between the two compared elements. The similarity scorings are added to *Match* if they are greater than or equal to a chosen threshold *t*. The latter is a real number between 0 and 1 that defines the strictness of the matching process.

The similarity between any two elements e_1 and e_2 is calculated based on their syntactic ($S_{Syntactic}$) and structural ($S_{Structural}$) similarity using the following formula:

$$\frac{S_{Syntactic} + S_{Structural}}{1 + S_{Syntactic}}$$

$S_{Syntactic}$ is best value obtained from Longest Common Substring (LCS) and n-gram [7] algorithms (n=2 in the proposed approach) where the computation is case-insensitive and all noise characters are normalized before the calculation. This is to overcome the change of word order, and the length of strings issues [18]. For example, *Professor* and *TeachingStaff* are 0.091 similar (*LCS* gives 0.091 while *2-gram* gives 0).

$S_{Structural}$ is calculated using the following formula:

$$\frac{2 \times sum}{sum + count}$$

Where *sum* is obtained by cumulating the syntactic similarities (or 0 or 1) between the compatible items of e_1 and e_2 and the elements associated with them, and *count* is the number of items/elements used in the calculation of *sum*. $S_{Structural}$ is not dependent on the order of the elements / items used in the calculation. Table 1 illustrates how the overall similarity between the operations *New* and *Add* of the classes *Professor* and *TeachingStaff* is computed.

Table 1. Similarity computation example

Aspect	Similarity
Classes	0.091
Visibility	1
Accessed Attributes	0.572
Inputs	0.75
Outputs	N/A
Preconditions	N/A
Postconditions	0.698
$S_{Syntactic}$	0
$S_{Structural}$	0.623
Overall similarity	0.623

During the computation of $S_{Structural}$, attribute names (as well as inputs and outputs) are replaced by their respective type when processing predicates (init, invariants, preconditions, and postconditions). The reason behind this 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 structurally similar elements. For example, even if the names of operations *New* and *Add* are not similar at all, their overall similarity is 0.623. Moreover, if two operations are matched, their parameters (inputs/outputs) are matched according to the similarities of the attributes they manipulate. Parameters with same names and types are not matched unless they manipulate matched attributes.

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 (i.e. Mutual-Enforcing-Relationship). In other word, classes are similar if they have similar elements and elements are similar if they are contained in similar classes. Finally, the best match is taken in case a specification's element is matched to more than one element.

3 DIFFERENCING SPECIFICATIONS

Given a set S of all the specifications, differencing between two specifications S_1 and S_2 is the process of identifying the exact set of operations (transformations) that allow obtaining S_2 from S_1.

The systematic identification of the exact differences that exist between two specifications requires a formal definition of the change operations involved. An algorithm to precisely compute this change can then be developed. Table 2 shows the proposed operations defining a difference between any two given specifications.

Table 2: Operations for differencing specifications

Operation	Effect
insertNode(e, t)	Inserts a new node *e* where *t* is the node's type. *t={Class, Variable, Operation, Predicate}*.
setNodeProperty(e, p, v)	Assigns for the 1st time a value *v* to the property *p* of the element *e*.
insertLink(k, e_1, e_2 , t)	Creates and inserts a new link *k* between the elements e_1 and e_2 where *t* is the link's type, *t={aggregated_by, derived_from, associated_with, declared_in, used_by}*.
deleteLink(k)	Removes the link *k*.
deleteAllLink(e)	Removes all *links* and *references* associated with the element *e*.
deleteNode(e)	Removes the node *e*.
Rename(e, oldname ,newname)	Renames the element *e* (named *oldname*) with *newname* and updates (with *newname*) all *references* made to *e* in the specification.
Modify(e, p, v_1, v_2)	Modifies the content of *e* by changing the values of a set of properties *p* (excluding the name) whose values are in v_1 with a set of new values v_2.

In addition to the precise and accurate representation of a difference between two given specifications, the above operations could also be used to represent the process of creating specifications. Moreover, the effect of deltas' operations can be inversed to obtain the old version of a specification. This is enabled by keeping track of old and new values (e.g. *Rename* and *Modify*), and the complementarities that exists between insertion and deletion operations, i.e. to revert the insertion of an element, we only need to delete it and vise versa.

The insertion of a node is concerned about four major meta-classes: *Class*, *Variable*, *Operation*, and *Predicate*. In case of OO formal specifications, the *Variable* meta-class has three sub-classes. They are the *Attribute* (global and state attributes) of a class, the *Input* and the *Output* of an operation. Moreover, the *Predicate* meta-class has four sub-classes. They are *Invariant*, *Initialization*, *Precondition* and *Postcondition*.

Renaming a specification's element requires updating all references made to it with its new name. For example, if a variable has been renamed, this name change is propagated to all elements that refer to this variable such as initialization, invariant, and pre (post) condition predicates. The same rule is applied when renaming classes and operations. Removing a specification's element requires removing its associated links, and all the references made to it as well (*deleteAllLink* operation). Furthermore, the operation *Modify* applies to both specifications' elements and links where a link's type (*p*) could be changed from v_1 to v_2. This reduces the number of operations in a delta by avoiding the removal of a link typed v_1 and the insertion of a link typed v_2. Table 3 highlights the different attributes of the meta-classes representing specifications' elements:

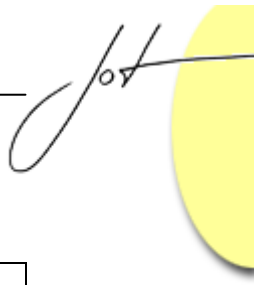


Table 3: The attributes of specifications' elements

Element	Attributes
Class	- name: the class' name
Variable	- name: the variable's name - data_type: is in the types supported by the formal language including class names. - visibility is in {yes, no, n/a}
Operation	- name: the operation's name - changes: is in {{some variable}, empty-set} - visibility is in {yes, no}
Predicate	- value: the predicate content is a set of String - visibility is in {yes, no, n/a}
Link	- type: the link's type is in {*aggregated_by, derived_from, associated_with, declared_in, used_by*}

Most of the attributes of table 3 are self-explanatory. However, there is a need to highlight the attributes *visibility* and *changes*. *Visibility* is similar to *public;* it applies to operations, some variables as well as some predicates. In case the visibility attribute is not applicable, the proposed value used is "n/a", such as in the case of inputs and outputs as well pre and post conditions. If an element needs to be visible outside the class the value "yes" is used otherwise "no" is used. The default visibility in Object-Z is "no", i.e. anything that needs to be visible outside the class has to be explicitly included in the visibility list. The *changes* attribute contains a set of variables that are changed by an operation. In case of a query operation, i.e. an operation that does not change the value(s) of the class variables it manipulates, the *changes* attribute is "empty".

Differencing is concerned about four categories of change. The insertion of elements / links, the modification of elements' contents, the modification of links' types, the deletion of elements / links, and the moving of elements. Using accurate matching results, differences between specifications can be precisely computed. Matched elements with different content are updates, matched elements with different links shows adding/removal of links and unmatched elements show adding/removal of elements.

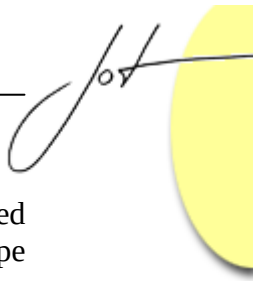
The difference between two given specifications is produced using the following algorithm.

```
Input: Specifications S1=<N1, L1> and S2=<N2, L2> where N1 (N2) is a set of nodes and
L1 (L2) is a set of links and Match containing the similar elements of S1 and S2.
Output: A set delta storing the difference
1.    delta=∅
2.    for all nodes n in N1 that are not in the domain of Match{
3.      delta = delta ∪ {deleteAllLink(n), deleteNode(n)}       }
4.    for all nodes m in N2 that are not in the domain of Match{
5.      delta = delta ∪ {(insertNode(m)}
6.      for all properties p of m with a value v{
7.        delta = delta ∪ {(setNodeProperty(m, p, v)}    }
8.      for all links k in L2 with m as source OR target{
9.        delta = delta ∪ {insertLink(k, k.source, k.target, k.type)}    }
10.   }
11.   for all x=(e1,e2,type) in dom(Match) do {
12.     if (e1.name≠e2.name) then delta = delta ∪ { Rename(e1, e1.name, e2.name) }
13.     if Match(x)<1 then {
14.       P={∀ p: Property. p ≠ name ∧ e1.p ≠ e2.p}
15.       if P≠∅ {
16.         V1={∀ p ∈ P. e1.p}
17.         V2={∀ p ∈ P. e2.p}
18.         delta = delta ∪ { Modify(e1, P, V1, V2) } }
19.       L1={∀ a: Element, ∃ k: Link. k.source=a ∧ k.target=e1}
20.       L2={∀ b: Element, ∃ k: Link. k.source=b ∧ k.target=e2}
21.       if (#L2 > #L1){
22.         K={k: Link, ∃ b ∈ {L2 \ L1}, k.source=b ∧ k.target=e2}
23.         for all link k in K do {
24.           delta = delta ∪ {insertLink(k, k.source, k.target, k.type)} }
25.       }
26.       else {
27.         K={∀ k: Link, ∃ a ∈ {L1 \ L2}, k.souce=a ∧ k.target=e1}
28.         for all links k in K do {
29.           delta = delta ∪ {deleteLink(k)}    }
30.       }
31.     }
32.   }
33.   return delta
```

Figure 2: Differencing algorithm

Given two specifications S_1 and S_2 represented by sets of nodes (N_1 and N_2) and sets of links (L_1 and L_2), the algorithm generates the exact set of operations (*delta*) that allow obtaining S_2 from S_1. The algorithm starts by analyzing the unmatched elements of the two specifications. The unmatched elements of S_1 are added to *delta* as being *deleted* (lines 2-3). In this case, the nodes as well as its associated links are deleted. The unmatched elements of S_2 are added to *delta* as being *newly inserted* elements (lines 4-5). Thus, all their associated properties and links are also added to *delta* (lines 6-9). The matched elements with different names are added to *delta* as renames (lines 11-12). For these elements, if they are not exact matches (i.e. similarity scoring < 1) then there is a possibility that their contents (other than names) have been modified, new links have been attached to them or that some of their links have been removed. The algorithm addresses this by detecting the changed properties other than names and adding them to *delta* (line 13-18). It also detects any new inserted links to them (lines 19-24) and any removed links (lines 26-29) and adds the changes to *delta*.

An important goal of differencing is to to have a mechanism that detects moved elements based on the modifications made in a delta. A potential moved element is a



Variable, an *Operation* or a *Class*. A *Variable* or an *Operation* is moved if a new added link k_2 of type "declared_in" connects it to a new class *B* and a link k_1 of the same type with an old class *A* is removed. A *Class* is moved if a new added link k_2 of type "derived_from", "aggregated_by" or "associated_with" connects it to a new class *B* and a link k_1 of the same type with an old class *A* is removed. A function *updateMoving* is used to perform the above verification; it accepts a *delta* containing a list of operations and a specification element *E* as parameters and returns an object containing the two elements representing the old and new link ends or a "null" object if no moving has taken place. Formally, this verification can be written as:

$$\exists k_1, k_2, E, A, B, t \text{ in } \{declared\_in, derived\_from, aggregated\_by, associated\_with\}, \{insertLink(k_2, E, B, t), deleteLink(k_1)\} \in delta: k_2.type = k_1.type \wedge k_2.source = k_1.source = E \wedge k_2.target \neq k_1.target$$

Let N_x be a reference to every specification elements named x and K_i (i=1...n) the new links inserted (if any). Table 4 shows the differences between the the class *Professor* and the classes *Academicien* and *TeachingStaff* of figure 1.

Table 4: Results of Differencing

Professor → Academicien	Rename($N_{Professor}$, "Professor", "Academician") deleteAllLink($N_{Affiliate}$) deleteNode($N_{Affiliate}$) Modify(N_{New}, changes, "{Id,Name,Expertise}", "{Id,Name,Expertise,Faculty}") insertLink(K_1, $N_{Faculty}$, N_{New}, "used_by") insertLink(K_2, $N_{f?}$, N_{New}, "declared_in") Modify($N_{postNew}$, value,"{Id'=i?,Name'=n?,Expertise'=e?}", "{Id'=i?,Name'=n?,Expertise'=e?,Faculty'=f?}")
Professor → TeachingStaff	Rename($N_{Professor}$, "Professor", "TeachingStaff") deleteAllLinks($N_{Expertise}$) deleteNode($N_{Expertise}$) Rename($N_{Faculty}$, "Faculty", "Institution") Rename(N_{New}, "New", "Add") Modify(N_{New}, changes, "{Id,Name,Expertise}", "{Id,Name}") deleteAllLinks($N_{e?}$) deleteNode($N_{e?}$) Modify($N_{postNew}$, value, "{Id'=i?,Name'=n?,Expertise'=e?}", "{Id'=i?,Name'=n?}") Rename($N_{Affiliate}$, "Affiliate", "Join")

The operations shown in table 4 represent the natural set of transformations a domain expert can find when comparing the studied classes manually. Although producing differences manually is tedious, they form the basis of validating a differencing approach. The latter should produce results that are as close as possible to deltas that are produced manually by a domain expert.

4 EMPIRICAL EVALUATION

A differencing approach must be efficient and should provide results that are precise and accurate. Precision is ensured by using primitive operations for differencing, acting on one specifications' element at a time, and containing traceability information allowing the reversal of a delta's effect. Accuracy can be measured by comparing the results produced

by the differencing algorithm to the operations that represent the kind of changes a human will intuitively find when he/she compares the specifications.

A Java prototype tool has been developed to help validating the proposed approach. It includes a component that parses specifications into graphs. Currently, only Object-Z specifications are supported. In addition, a differencing component is used to compare any two specifications represented as graphs using their computed matching results to produce a set of operations differentiating them.

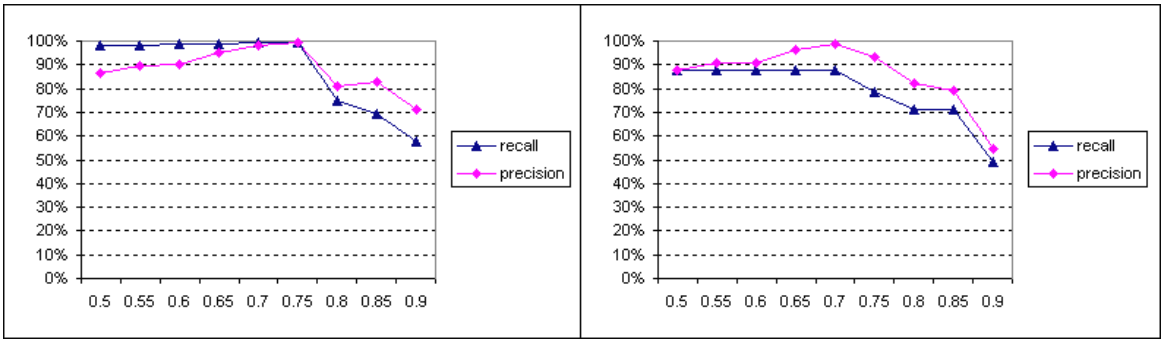
Several medium sized specifications have been used in the experiments. The first author of this paper created the specifications and two different domain experts were in charge of reviewing the created specifications and make any amendments to them according to requirements they think should be taken into consideration. These case studies include among others: a *university management system*, a *hotel management system*, an *online purchase system*, and a *project tracking and monitoring system*. The validation process involves comparing the results produced using the differencing approach with differences created manually by a third domain expert. Table 5 shows a summary of these differences.

Table 5: Details of the experiments

	V	V_1			V_2		
	Elements & Links	Insertions	Deletions	Modifications	Insertions	Deletions	Modifications
Case 1	77	13	5	20	4	22	11
Case 2	131	0	12	33	7	16	18
Case 3	216	0	35	27	20	15	23
Case 4	183	5	19	4	11	7	25
Total	608	173			179		

The combined base specifications (V) contain a total number of 608 elements and links. The first versions of the specifications (V_1) were obtained after performing 173 delta operations and the second versions (V_2) were obtained through 179 delta operations made to the base specifications respectively.

The differencing approach was validated through the number of correct operations produced (positives), the number of all operations produced (positives and false positives), the number of correct operations missed (negatives), and the total number of correct operations (as shown in Table 5). Precision and recall metrics were used in the evaluation. Precision measures quality and is the ratio of the number of correct operations produced to the total number of operations produced. Recall measures coverage and is the ratio of the correct operations produced to the total number of correct operations. Figure 3 shows the experimental results obtained for similarity thresholds ranging from 0.5 to 0.9.



(a) (b)

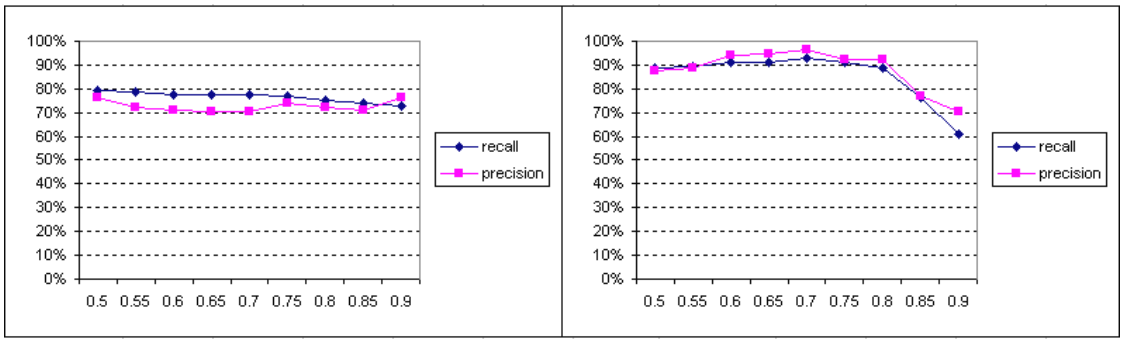
Figure 3: Experimental results

The combined results obtained through differencing the base specification V and the first version V_1 for all case studies used in the experiments are shown in figure 3 (a). A very good recall (98%-99%) combined with a good precision (86%-95%) were obtained for thresholds raging from 0.5 to 0.65. Moreover, a very good recall (99%) combined with a very good precision (98%-99%) were obtained for threshold ranging from 0.7 to 0.75. Furthermore, an acceptable recall (69%-75%) combined with good precision (81%-83%) were obtained for thresholds ranging from 0.8 to 0.85. For a threshold equal to 0.9, only 100 correct operations were obtained out of the 140 operations produced compared to the 173 correct operations, which is indicated by a considerable drop of the approach's recall (58%), and a drop in its precision (71%).

Figure 3-(b) shows the combined results obtained through differencing the base specification V and the second version V_2 for all case studies used in the experiments. A good recall (79%-88%) combined with a good precision (88%-99%) were obtained for thresholds raging from 0.5 to 0.75. For thresholds ranging from 0.8 to 0.85, the recall and precision obtained were lower, 71% and 79%-82% respectively. A high threshold of 0.9 resulted in a sharp drop in the approach's precision (54%) and recall (49%). The average precsion and recall obtained for the combined experiments (in (a) and (b)) were 87% and 83% respectively.

For a reference threshold of 0.7, the approach's recall and precision for the combined experiments (in (a) and (b)) were 93% and 99% respectively. The performance was almost the same for thresholds ranging from 0.6 to 0.65. Consequently, the differencing approach performed the best for thresholds ranging from 0.6 to 0.7 as the average recall and precision obtained were 92% and 97% respectively.

To study the impact of the distribution of the modifications made on the performance of the differencing approach, we have edited the specifications versions in such a way that: (a) most of the modifications made are concentrated on a small number of specifications' elements, and (b) the modifications are distributed fairly between the elements of the specifications. Figure 4 shows the combined results obtained in terms of precision and recall.



(a) (b)

Figure 4: Results of controlled experiments

The overall results obtained showed that precsion and recall of the differencing approach were consistent when the modifications made are concentrated mainly on a small number of elements (figure 4 (a)). The precision and recall obtained were between 70% and 79%, with an average precision of 73% and an average recall of 77%. When the modifications made are well distributed across the specifications' elements (figure 4 (b)), the results obtained were slightly better compared to the results of the uncontrolled experiments. The average precision and recall obtained were 88% and 86% respectively. Thus, if the modifications made fall under categorie (a), there is no considerable impact on the performance of the differencing approach. However, if they fall under category (b), the differencing results produced are slightly better and closer to the kind of changes a human will intuitively find when he/she compares the specifications manually. This is because specifications' elements are more precisely matched when they undergo small change. This translates into a better performance for the differencing approach.

The proposed approach scales up well in terms of efficiency (performance and memory usage) as the size of the specifications increases. This is due to three main reasons. Firstly, the approach used to detect the similarities between specifications has an acceptable complexity bounded by $O(nm)$ where n and m are the number of elements of the specifications. In addition, only compatible elements are compared, i.e. classes with classes, variables with variables, etc. Thus, the actual number of comparison is far smaller than $n*m$. Secondly, during delta calculation only the difference between the specifications is calculated and stored, which leads to a more efficient memory usage. Finally, the proposed approach is operation-based which leads to a better performance when merging the differences because the operations contained in the deltas are compared rather than comparing the input specifications themselves. Knowing that the number of operations a delta can have is statistically smaller than the number of specifications' elements.

5 RELATED WORK

In [20], a differencing algorithm is proposed to detect the structural changes between the designs of subsequent versions of OO software. The algorithm reports the differences between them in terms of additions / removals, moves, and renaming of program elements such as packages and classes. The differencing algorithm computes an overall similarity based on name and structure similarity metrics. The proposed algorithm assumes that enough design entities remain the "same" between the two consecutive versions of the system. The latter assumption is weak as there is no guarantee that the developers of the new version of the system do not make too many modifications. The experimental results obtained reported limitations in detecting moved fields and methods. Moreover, any mistakenly identified renaming or moving of an entity is propagated to the class or the interface that contains it, and the latter will be reported as changed as well.

In [19] an algorithm is proposed to detect changes in XML documents. As a mean to improve change results, unordered tree representation of the analyzed models were used. The matching part of the approach uses nodes signatures and prevents matching child nodes with different ancestors. This restriction affects the change detection by limiting the recognition of moved nodes. The experimental results obtained showed a slow running time while leading to a good accuracy. Similar differencing algorithms were proposed in [1] and [3] to deal with different kind of documents namely OO programs and UML models respectively.

In [4], an approach for fine-grained source code change extraction is proposed. The approach processes programs as trees and uses a combination of proven similarity metrics and measures to improve the detection of matching between versions of a program. These matching are used to produce tree edit operations comprising insertions, deletions, alignments, moves and updates. The approach has been empirically evaluated using data extracted from open source case studies. The results obtained showed a mean error rate of 34%, which is an improvement from the 79% obtained with the algorithm [3] based on which their work was based.

Finally, in [15] an approach is proposed to perform a change impact analysis on OO programs. Source code edit operations are transformed into a set of atomic changes. Eight categories of atomic changes were defined for the fields, the methods, and the classes of an OO program. Given a program and a set of test drivers exercising a program's methods, the approach formally defines the impact of the change undergone by the program on the behavior of the test drivers. The drawback of the proposed approach lies in the assumption that the edit operations performed on the program are obtained directly through an Integrated Development Environment (IDE). This may apply to some IDEs, but in general, this is not applicable. For example, the proposed change impact analysis cannot be applied to a Java program that has been edited using a simple text editor. Thus, the approach lacks a proper differencing mechanism capable of computing the change undergone by the studied OO programs.

6 CONCLUSION AND FUTURE WORK

The paper discussed an approach for differencing OO formal specifications. The proposed approach processes specifications as graphs unlike existing approaches that are based on tree data structures. Thus, a better generality is achieved. The proposed approach computes matching results using proven similarity metrics combining syntactic and structural information about the compared specifications' elements, and is not dependent on the order of elements (nodes). The change operations produced were formally defined as a set of primitive operations whose effects can be reversed because of the traceability information they contain. The proposed approach incorporates also a mechanism capable of detecting moved elements. The approach was empirically validated, an average precision of 87% and an average recall of 83% were obtained for the combined experiments made. The differencing approach proposed in this paper is part of a project that aims at merging OO formal specifications resulting from collaboration. For future work, the emphasis will be on combining the deltas to merge specifications, and detecting and resolving merge conflicts. Finally, providing means to compress the result of the differencing approach could be explored as it leads to a better efficiency, and it is important to run more experiments on the differencing approach using larger specifications.

REFERENCES

[1] Apiwattanapong, P., Orso, N. & Harrold, M.J. (2007). JDiff: A Differencing Technique and Tool for Object-Oriented Programs. Automated Software Engineering, 14(1), 3-36.

[2] Boronat, A., Carsi, J.A., Ramos, I. & Letelier, P. (2007). Formal Model Merging Applied to Class Diagram Integration. Electronic Notes in Theoretical Computer Science, 166 (1), 5–26.

[3] Chawathe, S. S., Rajaram, A., Garcia-Molina, H. (1996). Change Detection in Hierarchically Structured Information. Proceedings of ACM Sigmod Int'l Conf. on Management of Data, pp.493-504.

[4] Fluri, B., Wursch, M., Pinzger, M. and Gall, H. C (2007). Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. IEEE Transactions of Software Engineering, 33(11), 725-743.

[5] Fortsch, S. & Westfechtel, B. (2007). Differencing and merging of software diagrams - state of the art and challenges. Proceedings of Int'l Conf. on Software and Data Technologies, pp. 90-99.

[6] Godfrey, M. W. & Zou, L. (2005). Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. IEEE Transactions on Software Engineering, 31 (2), 166-181.

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

[8] Hinchey, M. G. (2008). Industrial-Strength Formal Methods in Practice, Springer.

[9] Ignat, C. L. & Norrie, M. C. (2004). Operation-based versus State-based Merging in Asynchronous Graphical Collaborative Editing. 6th Int'l Workshop on Collaborative Editing, IEEE Distributed Systems Online.

[10] Kelter, U., Wehren, J. & Niere, J. (2005). A Generic Difference Algorithm for UML Models, Proceedings of Software Engineering Conference, pp. 105-116.

[11] Kontonya, G. & Sommerville, I. (2002). Requirements Engineering Process and Techniques, John Wiley and Sons.

[12] Lippe, E. & Oosterom, N. V. (1992). Operation-based Merging. ACM SIGSOFT Software Engineering Notes, 17 (5), 78-87.

[13] Mehra, A., Grundy, J. & Hosking, J. A. (2005). Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. Proceedings of Int'l Conf. on Automated Software Engineering, pp 204-213.

[14] Mens, T. (2002). A State of the Art Survey on Software Merging. IEEE Transactions on Software Engineering, 28 (5), 449-462.

[15] Ryder, B. G., Tip, F. (2001). Change impact analysis for object-oriented programs. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, pp. 46-53.

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

[17] Sriplakich, P., Blanc, X. & Gervais, M. P. (2008). Collaborative Software Engineering on Large-Scale Models: Requirements and Experience in ModelBus. Proceedings of SAC'08 Conference, pp. 674-681.

[18] Taibi, F., Abbou, F. M. & Alam, M. D. (2008). A Matching Approach for Object-Oriented Formal Specifications. Journal of Object Technology, 7(8), 139-153.

[19] Wang, Y. (2003). X-Diff: An Efficient Change Detection Algorithm for XML Documents. Proceeding of 19th Int'l Conf. on Data Engineering, pp. 519-530.

[20] Xing, Z. & Stroulia, E. (2007). Differencing logical UML models, Journal of Automated Software Engineering, 14(2), 215-259.

About the authors

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

Dr. Junaidi Abdullah is a lecturer at the Faculty of Information Technology of Multimedia University. His research interests include computer vision, image processing and augmented reality. He can be reached at junaidi@mmu.edu.my.