J. Software Engineering & Applications, 2010, 3: 460-471
doi:10.4236/jsea.2010.35052 Published Online May 2010 (http://www.SciRP.org/journal/jsea)

A Conflicts Detection Approach for Merging Formal Specification Views

Fathi Taibi[1], Fouad Mohammed Abbou[2], Md. Jahangir Alam[2]

[1]University of Tun Abdul Razak, Malaysia; [2]Multimedia University, Malaysia.
Email: 1taibi@unitar.edu.my, {fouad, md.jahangir.alam}@mmu.edu.my

Received April 20th, 2009; revised June 2nd, 2009; accepted June 10th, 2009.

ABSTRACT

Specifying software requirements is an important, complicated and error prone task. It involves the collaboration of several people specifying requirements that are gathered through several stakeholders. During this process, developers working in parallel introduce and make modifications to requirements until reaching a specification that satisfies the stakeholders' requirements. Merge conflicts are inevitable when integrating the modifications made by different developers to a shared specification. Thus, detecting and resolving these conflicts is critical to ensure a consistent resulting specification. A conflicts detection approach for merging Object-Oriented formal specifications is proposed in this paper. Conflicts are classified, formally defined and detected based on the results of a proposed differencing algorithm. The proposed approach has been empirically evaluated, and the experimental results are discussed in this paper.

Keywords: *Formal Specification*, *Object-Oriented*, *Collaboration*, *Merge Conflicts*, *Consistency*

1. Introduction

The development of large-scale software systems requires the collaboration [1] of hundreds of developers working on different aspects of the same system. Often, this leads to the creation of different but related documents. These documents (views) could be in the form of design models, software specifications, source code, etc. During a particular collaborative activity, a resulting local view needs to be merged [2] with the version of the document available in a shared repository. The latter shared document encloses all the modifications made locally and checked (integrated) into the repository at that point in time. A merging approach must integrate the changes made and must ensure that the merging result is consistent by detecting and resolving merge conflicts [3].

Specifying software requirements is an important, complicated and error prone task that involves the collaboration of several peopled specifying requirements that are gathered 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 [4].

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 [5] offer a better alternative because of their precise and accurate nature. Object-Oriented (OO) formal methods, such as Object-Z [6], 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, making them suitable to be used when developing specifications collaboratively why they can be manipulated systematically.

Conflicts detection requires the calculation of the delta(s) representing the modifications made locally compared to a shared view. Analyzing these deltas allows the detection of conflicts before integrating their content with the shared view. Rules could be formally defined for these conflicts to uniformly detect different type of violations such as those concerned with the loss of update, the well formedness of the view, the avoidance of all kind of redundancies, etc.

Efficient merging frameworks support collaboration and distributed development, and when used at an early phase of the software development such as when specifying software requirements, they enable detecting conflicts that will cost higher to detect and resolve during later stages of development.

Employing unique identifiers for the elements of the merged documents, such as in [7], introduce 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 [8], merging should not rely on elements' unique identifiers. Thus, a differencing approach is needed to produce a list of the created/deleted and modified elements as well as the created/modified relationships (or links) between the elements of the specifications. Differencing two specifications should be based their computed similarities (matches). The similarities between the elements of two specifications could be calculated accurately based on syntactic as well as structural similarity.

Several existing work on model differencing and version control such as CVS [9] adopts line-based (textual) management. Textual merging tools have been used to some extend in industry, and in [10] it has been reported that around 90% of the changed file could be merged without any problem. However, the remaining changes cannot be merged automatically because there is no consideration to the syntactic or semantic information of the files. In order to manage the changes of specifications there is a need to work at the granularity of a logical unit component such as a class rather than at the granularity of a line. Moreover, textual merging can only detect very basic conflicts, as it does not take into account the specific structure of the processed documents. Furthermore, it gives rise to unimportant conflicts [11] such as code comments that have been modified, line breaks, etc. Thus, transforming specifications into a textual format used by existing tools cannot solve the problem of merging specifications.

Most of existing merging approaches process the manipulated documents as trees, which is restrictive and not applicable to a large number of documents including software requirements specifications. Moreover, most of the surveyed approaches are inadequate for merging specifications due to their limitations in uniformly defining conflicts, and accurately detecting and resolving them. The domain independent approaches surveyed lack accuracy when used to merge specifications, and to our knowledge, there is no existing merging approach intended specifically for Object-Oriented formal specifications.

In this paper, an approach is proposed to detect and resolve conflicts when merging OO formal specifications. The approach comprises three parts. The first part consists of comparing specifications to produce deltas differentiating them. The second part consist of integrating (merging) the deltas into a shared specification and the third part consists of checking the deltas against defined consistency rules to detect and resolve any consistency violations that might arise during merging. Collaborative formal specification is discussed in the next section. This is followed by proposing an algorithm for differencing OO formal specifications. After that, an approach for detecting and resolving merge conflicts is discussed. Then, the proposed approach is empirically evaluated. This is followed by discussing related work and the last section concludes the paper and discusses future work.

2. Collaborative Development of Formal Specifications

Software development is a collaborative activity as it involves several people working on different aspects of the same software project. Most often, collaboration is the key to the success of a software project. During the specification of software requirements, developers working in parallel add, remove, and modify requirements until reaching a description of the system (or subsystem) that satisfies the stakeholders' requirements. This collaborative nature raises the needs for frameworks to support the merging of software specifications.

Asynchronous collaboration allows members of a group to modify copies of a shared specification in 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. In such environments, three basic operations are applied on a shared repository of specifications: check-out, modify, and check-in. The check-out operation consists of importing a copy of the latest version (for example at time t_0) of a shared specification (S_{Base}) from the repository in order to perform some modifications on it. These modifications represent new requirements that have yet to be specified or different views on the requirements that have already been specified. The modifications made lead to the creation of a new version S_{Local} of the shared specification S_{Base}. Check-out is not applicable to the first developer creating the first version of a given specification and checking it into the repository. However, later on, he/she can perform a check-out operation to make any new changes if needed. In case at time t_1 ($t_1 > t_0$), the shared specification has evolved into a new version S_{Rep} after undergoing some changes made by a different developer. The check-in operation consists of merging the local and shared versions of the specification in case of two-way merging. In case of three-way merging, the modifications made locally need to be adjusted according to those who have already been integrated in S_{Rep}, *i.e.* using two specifications (S_{Local} and S_{Rep}) and their ancestor (S_{Base}) in the merge. Integrating all the modifications made with the base specification is possible after that. The following figure illustrates the basic operations of this collaborative environment:

A developer 'X' imports (check-out) a shared specification (S_{Base}) from a Specification repository. He/she

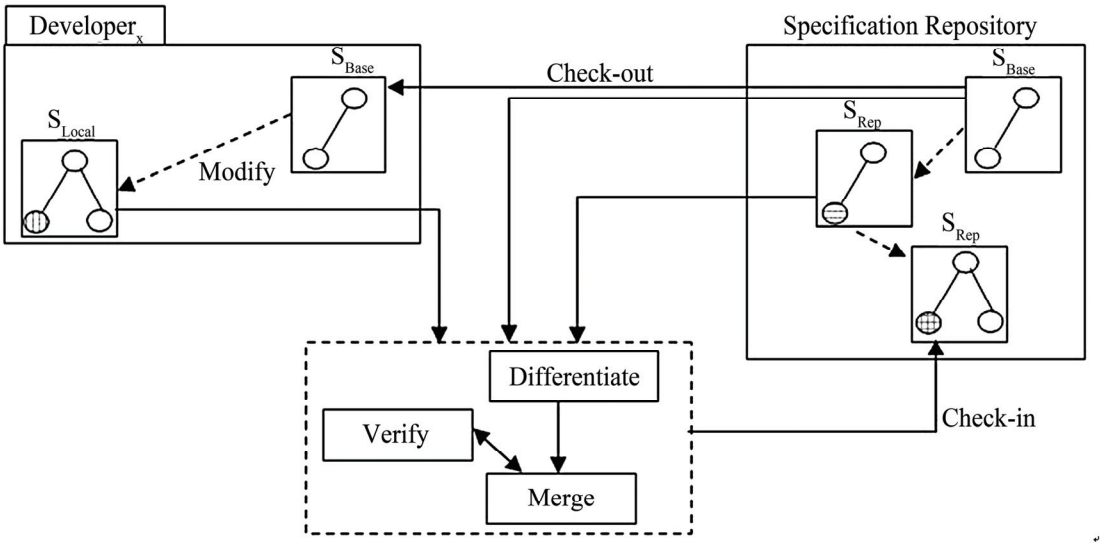

Figure 1. The operations supporting collaboration

performs some modifications on it locally. These modifications lead to the creation of a new version (S_{Local}) of the specification. For X to check-in S_{Local} into the specification repository, there is a need to discover the exact modifications (delta) made. The operations contained in this delta allow detecting the conflicts that might arise during the merge. In case of three-way merging, there is a need to identify the exact modifications made locally and those that have evolved S_{Base} into S_{Rep} because of some parallel modifications made and integrated by a different developer 'Y'. In addition to the latest version of the shared specification, a history of all the deltas applied to it is also stored in the repository. Thus, re-obtaining S_{Base} from which S_{Rep} originates is achieved by reversing the effect of the last integrated delta. Using S_{Base} and the two deltas, a three-way merging could take place where conflicts caused by the parallel modifications need to be detected and resolved before integrating them.

The proposed framework incorporates three approaches to perform the required tasks. The first approach "Differentiate" is intended to differentiate between the to-be-merged specifications. The differentiation process involves the production of deltas containing the exact modifications (operations) made. The latter deltas are obtained using the computed similarities that exist between the specifications' elements. Merging based on differencing is referred to as operation-based merging [11] and is efficient in case of large models as the number of operations that transform a version into another one is statistically smaller than the number of model elements. Furthermore, it provides a better platform for conflicts detection and resolution. The second approach "Merge" is intended to merge the modifications made with the shared specification. The specification obtained through this process must be consistent. Thus, the third approach "Verify" is intended to detect and resolve merge conflicts. The differencing and verifying approaches are discussed in detail in sections 3 and 4.

3. Differencing Object-Oriented Formal Specifications

Given a basic 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. 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.

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 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 while the operations *New* and *Affiliate* have been renamed as *Add* and *Join* respectively. In addition, the operation *New* has been

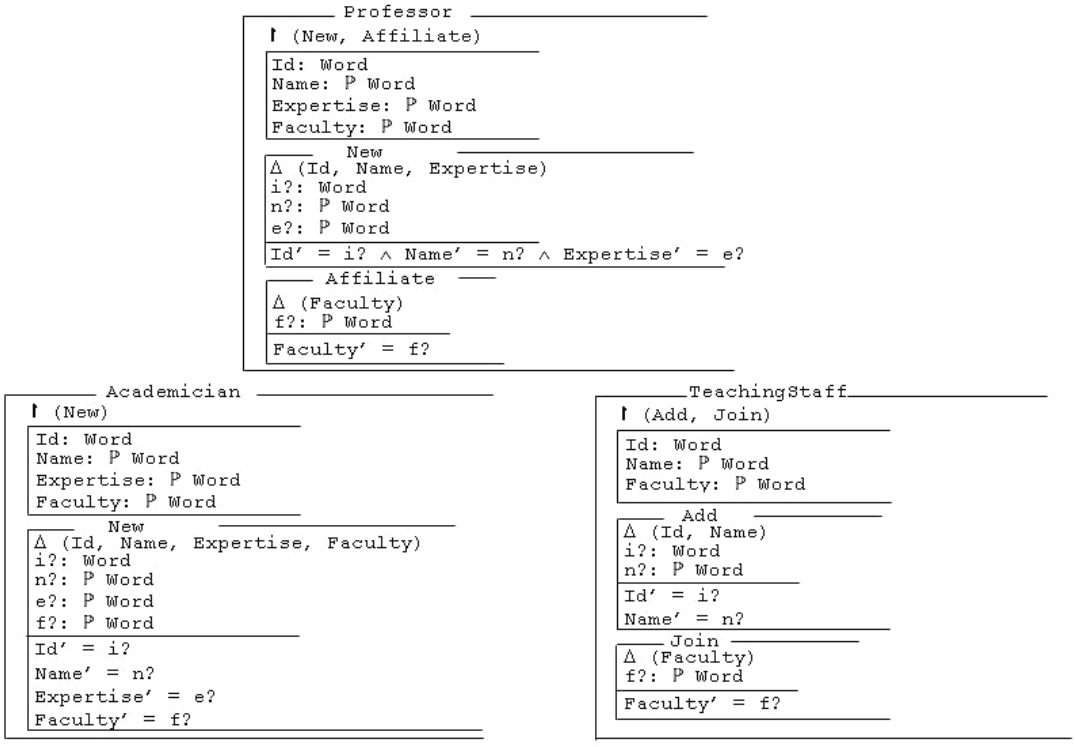

Figure 2. Three versions of an Object-Z class

modified by removing the part dealing with the deleted attribute *Expertise*.

The systematic identification of the exact differences that exist between the class *Professor* and the classes *Academician* and *TeachingStaff* respectively requires a formal definition of the change undergone by a specification. An algorithm to precisely compute this change can then be developed. **Table 1** shows the proposed operations defining a difference between any two given specifications.

In addition to the precise and accurate representation of a difference between two given specifications, the above operations could also be used to represent specifications' creation process itself. Moreover, the effect of a delta's 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 complementarity 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*.

Table 1. 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 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.

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 2** highlights the different attributes of the meta-classes representing specifications' elements:

Most of the attributes of **Table 2** 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 three classes of change. The insertion of elements/links, the modification of elements' contents, the modification of links' types, and the deletion of elements/links. We propose a differencing algorithm that is not based on elements' identifiers but rather on matching results. 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 similarities that exist between specifications' elements are stored in a matching function:

$$\textbf{Match : ELEMENT} \times \textbf{ELEMENT} \times \text{TYPE} \rightarrow \text{R}$$

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. The latter is a real number between 0 and 1 that defines the strictness of the matching process [12].

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. The link's source and target are O and A respectively, and its type is "declared_in" as in defined in **Table 1**. The difference between two given specifications is produced using the following algorithm. Given two specifications S_1 and S_2 representing 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 transformation 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 2-4). In this case, the nodes as well as their associated links are deleted. The unmatched elements of S_2 are added to *delta* as being *newly inserted* elements. Thus, all their associated properties and links are also added to *delta* (lines 5-7). The matched elements with different names are added to *delta* as renames (lines 9-11). 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). It also detects any new inserted links to them (line 14) and any removed links (line 15) and adds the changes to *delta*.

Table 2. 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*}


```
1.  delta=Ø
2.  for all nodes n in N1 that are not in the domain of Match do
3.     add "delete" operation to delta removing the node n and all the links associated with it
4.  end for
5.  for all nodes m in N2 that are not in the domain of Match do
6.   add "insert" operation to delta inserting the node m, setting its properties, and inserting the links associated with it
7.  end for
8.  for all x=(e1,e2,type) in domain of Match do
9.    if (e1.name≠e2.name) then
10.    add"rename" operation to delta renaming e1 with e2.name
11.   end if
12.   if Match(x) < 1 then
13.    select the properties (other than name) of e2 with values different from e1, add "modify" operations to delta
14.    select all links to/from e2 with no matching link to/from e1, add "insert" operations to delta
15.    select all links to/from e1 with no matching link to/from e2, add "delete" operations to delta
16.   end if
17. end for
18. return delta
```

Figure 3. Differentiate algorithm

Low similarity thresholds lead to a high rate of false matches while being able to detect most of the correct matches. High thresholds lead to few or no false matches while producing a high rate of missed matches. In [12], empirical results have shown that a reasonable threshold value (around 0.7) produces the most balanced results, which leads to a more precise delta calculation.

4. Detection and Resolution of Merge Conflicts

Detection and resolution of conflicts are treated as two separated phases. This is to allow each one of them to be fined-tuned without an influence on the other one. Conflicts detection should be systematic while conflict resolution might require some user interaction. Given two deltas (*delta*$_1$ and *delta*$_2$) and a base specification from which both originate, conflicts detection in concerned about discovering conflicts that might arise when the modifications contained in the deltas are integrated with the base specification. The goal of the approach is to produce a conflict-free delta whose operations are the unification of *delta*$_1$ and *delta*$_2$. The operations contained in this delta are then applied to the base specification to perform the merge.

4.1 A Formal Definition of Conflicts

The goal of merging is to combine the modifications made and preserve their intensions. Conflicts should be resolved accordingly. **Tables 3** and **4** show a classification and a formal definition of the most frequent conflicts observed and their causes. In these tables, *p* refers to a property or a list of properties, *v* (v_i) refers to a value or

Table 3. Lost update conflicts

Conflict rule	How the confilict happens?
Rule 1: *modify-deleted-element*	∃ e, p, $v_1 \neq v_2$: Modify(e, p, v_1,v_2) and deleteNode(e) *An element e is modified in one delta while it is deleted in the other one.*
Rule 2: *modify-deleted-link*	∃ k, p="type", $v_1 \neq v_2$: Modify(k, p, v_1,v_2) and deleteLink(k) *The type of a link k is modified in one delta while it is deleted in the other one.*
Rule 3: *rename-deleted-element*	∃ e, $v_1 \neq v_2$: Rename(e, v_1,v_2) and deleteNode(e) *An element e is renamed in one delta while it is deleted in the other one.*
Rule 4: *concurrent-update*	∃ e, p, $v_1 \neq v_2 \neq v_3$: Modify(e, p, v_1,v_2) and Modify (e, p, v_1,v_3) *An element e undergoes different modifications in the two deltas.*
Rule 5: *concurrent -renaming*	∃ e, $v_1 \neq v_2 \neq v_3$: Rename(e, v_1,v_2) and Rename (e, v_1,v_3) *An element e undergoes different renaming in the two deltas.*
Rule 6: *modify-moved-element*	∃ e, p, $v_1 \neq v_2$: Modify(e, p, v_1,v_2) and Move(e) *An element e is modified in one delta while it is moved in the other one.*
Rule 7: *rename-moved-element*	∃ e, $v_1 \neq v_2$: Rename(e, v_1,v_2) and Move(e) *An element e is renamed in one delta while it is moved in the other one.*
Rule 8: *concurrent-moving*	∃ e, $e_1 \neq e_2 \neq e_3$: Move(e)=(e_1 , e_2) and Move(e)=(e_1, e_2) *An element e undergoes different moving in the two deltas.*
Rule 9: *move-deleted-element*	∃ e: Move(e) and deleteNode(e) *An element e is moved in one delta while it is deleted in the other one..*

Table 4. Structural conflicts

Conflict rule	How the confilict happens?
Rule 10: *modify-source-class*	$\exists$ A, A_1, p, $v_1 \neq v_2$, k, v: Modify(A, p, v_1,v_2) and insertLink(k, A, A_1, v) *An class A is modified in one delta while it is the source of a new link in the other one.*
Rule 11: *modify-target-class*	$\exists$ A, A_1, p, $v_1 \neq v_2$, k, v: Modify(A, p, v_1,v_2) and insertLink(k, A, A_1, v) *An class A is modified in one delta while it is the target of a new link in the other one.*
Rule 12: *link-without-source*	$\exists$ e, e_1, k, v: deleteNode(e) and insertLink(k, e, e_1, v) *An element e is removed in one delta while it is the source of a new link in the other one.*
Rule 13: *link-without-target*	$\exists$ e, e_1, k, v: deleteNode(e) and insertLink(k, e, e_1, v) *An element e is removed in one delta while it is the target of a new link in the other one.*
Rule 14: *double-containment*	$\exists$ k, e, e_1, v= "declared_in":insertLink (k, e, e_1, v) and ($\exists$ k_1:k_1.source=k.source $\wedge$ k_1.target$\neq$ k.target $\wedge$ k_1.type=k.type) *An element e is linked through a "declared_in" relation with an element e_1, in one of the deltas while it is linked through the same type of relation with another element in the base specification.*
Rule 15: *symmetric-link*	$\exists$ k, e, e_1,v in {"derived_from", "declared_in", "used_by"}: insertLink(k, e, e_1,v) and($\exists$ k_1: k_1.source=k.target $\wedge$ k_1.target=k.source $\wedge$ k_1.type=k.type) *An element e is linked through a "derived_from", "declared_in" or "used_by" relation with an element e_1, in one of the deltas while e_1 is linked to e through the same type of relation in the base specification.*
Rule 16: *cyclic-class-link*	$\exists$ A, A_1, k, v: insertLink(k, A_1, A, v) and ($\exists$ TransitiveClosure$_v$(A, A_1)) *An new link between two classes A_1 and A is inserted in one of the deltas while there a transitive closure between A and A_1 in the base specification.*
Rule 17: *redundant-link*	$\exists$ A, A_1, k, v: insertLink(k, A, A_1, v) and ($\exists$ TransitiveClosure$_v$ (A, A_1)) *An new link between two classes A and A_1 is inserted in one of the deltas while there a transitive closure between those classes in the base specification.*
Rule 18: *unwanted-reachability*	$\exists$ k, k_1, e, e_1, e_2, v: insertLink(k, e, e_1, v) and insertLink(k, e_1, e_2, v) *An new link between two elements e and e_1 is inserted in one delta while a new link of the same type is inserted in the other one conneting e_1 to an elememt e_2 leading to a transitive reachability between e and e_2.*
Rule 19: *redundant-element*	$\exists$ k, e, e_1, e_2, v= "declared_in": insertLink(k, e_2, e, v) and ($\exists$ k_1: k_1.source=e_1 $\wedge$ k_1.target=k.target $\wedge$ k_1.type=k.type $\wedge$ e_2 name=e_1.name) *An element e_2 is linked through a "declared_in"relation with an element e in one of the deltas while there is an elememt e_1, with the same name as e_2 that is linked with e through the same type of relation in the base specification.*
Rule 20 *double-definition*	$\exists$ e_1, $v_1 \neq v_2$:Rename(e_1, v_2,v_1) and ($\exists$ e:e.name=v_1) *An element e_1 is renamed in one of the deltas while the name is already being used in the base specification for a different element e.*

to a list of values, *e* (e_i) refers to specifications' elements, *A* (A_i) refers to classes, and *k* (k_i) refers to links between elements.

The conflicts have been classified under two categories.The first category is concerned about lost updates, which happens when the effect of the modifications made in one delta forbid those in the other one. Nine rules have been formally defined to allow the precise detection of this type of conflicts. The second category is concerned about the structural consistency of the specification obtained after integrating the modifications made in the delta(s), where eleven rules were formally defined. Structural consistency is a prerequisite to ensuring other forms of consistency such as those related to specifications' semantics. In fact, consistency checking, such as model checking in case of formal specifications, produces meaningful results only when applied to specifications that are well formed.

Several conflicts under the lost update category originate because of moving elements, thus, it is important 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. An operation *Move* 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:

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

Finally, in order to detect some structural conflicts such as *cyclic-class-link and redundant-link*, an operation $TransitiveClosure_v$ is used. Given any two classes A_1 and A_2, the operation is used to verify the existence (*True* or *False*) of a path (of more than two links) between A_1 and A_2 whose links are all of type v. This can be formally written as:

$\exists\, k_i$: i *in* $[1...n]$ *where* $n>=2$ *and* $\forall i$: $k_i.type=v$, $\exists\, B_j$:j *in* $[1...m]$ *where* $m=n-1$: $(k_1.source=A_1 \wedge k_1.target=B_1 \wedge k_2.source= B_1 \wedge k_2.target= B_2 \wedge \ldots \wedge k_n.source=B_{n-1} \wedge k_n.target=A_2)$

4.2 Conflicts Detection

The proposed approach accepts as input a base specification (S_{Base}) and two deltas ($delta_1$ and $delta_2$) representing the parallel modifications made. Based on the formal definitions proposed previously, the approach generates a list C containing the details of all the conflicts discovered. **Figure 4** shows the algorithm used to discover these conflicts.

The elements of $delta_1$ and $delta_2$ are traversed to discover conflicting operations according to pre-defined rules (e.g. rules 1 to 20 of **Tables 3** and **4**). In case such operations are found, an object containing the indexes of the operations causing the conflict in $delta_1$ and $delta_2$, the type of conflict, and a conflict resolution (if any) is added to the conflict list C (line 9). Unlike lost updates, which are caused by two operations, structural conflicts may originate because of one operation only (from one of the deltas) or two operations (from both deltas). Thus, the conflict object added to C in this case may include only one index identifying the operation causing the conflict and a "null" value is assigned for the second index (lines 2-3 and lines 6-7). At the end of this process, the conflict list C will be storing the details of all discovered conflicts.

It is important to note that the most frequent structural conflicts originate mainly because of the creation of new links and modifying (including renaming) the elements of a base specification, and that it is possible to reduce the number of conflicts detected by enforcing conflict rules only to a specific high-level granularity such as the *Class* level.

4.3 Resolving Conflicts

A Conflict resolution is a set of transformation applied to the delta(s) so that a conflict is resolved. For example in case of a *visibility* attribute undergoes different modifications, a resolution is to keep the most restrictive one. Most often, a resolution consists of dropping or altering one of the conflicting operations. Manual conflict resolution is a tedious, error prone and time-consuming process especially for large specifications. Moreover, the interpretation of conflicts can differ from one developer to another one. Thus, there is a need for an approach to support the systematic resolution of as many conflicts as possible. Interacting with developers only when several (or no) resolutions are possible and a choice need to be made.

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 5** shows the deltas and the conflict list C associated with the classes of **Figure 2**.

Four lost update conflicts were discovered through the application of the proposed conflicts detection approach. The first conflict is a *concurrent-renaming* originating because of two different renaming of the class *Professor*. A resolution to this conflict consists of dropping the renaming operation of $delta_2$. The second conflict is a *rename*

Input: A base specification S_{Base} and two deltas $delta_1$ and $delta_2$
Output: A conflict list C

```
1.  for all operations op1 in delta1 {
2.    if (effect of op1 on SBase causes a conflict) then
3.      C=C ∪ {New Conflict(indexOf(op1),0, getConflictType(),getResolution())}
4.  if delta2 ≠Ø then {
5.    for all operations op2 in delta2 {
6.    if (effect of op2 on SBase causes a conflict) then
7.      C=C ∪ {New Conflict(O, indexOf(op2), getConflictType(),getResolution())}
8.    if (combined effect of op1 and op2 on SBase causes a conflict) then
9.      C=C ∪ {New Conflict(indexOf(op1), indexOf(op2), getConflictType(),getResolution())}
10.   }
11. }
12.}
13. return C
```

Figure 4. Verify algorithm

Table 5. A conflict list of two deltas

$delta_1$	0) Rename($N_{Professor}$, "Professor", "Academician") 1) deleteAllLink($N_{Affiliate}$) 2) deleteNode($N_{Affiliate}$) 3) Modify(N_{New}, changes, "{Id,Name,Expertise}", "{Id,Name,Expertise,Faculty}") 4) insertLink(K_1, $N_{Faculty}$, N_{New}, "used_by") 5) insertLink(K_2, $N_{f?}$, N_{New}, "declared_in") 6) Modify($N_{postNew}$, value,"{Id'=i?, Name'=n?, Expertise'=e?}", "{Id'=i?, Name'=n?, Expertise'=e?, Faculty'=f?}")
$delta_2$	0) Rename($N_{Professor}$, "Professor", "TeachingStaff") 1) deleteAllLink($N_{Expertise}$) 2) deleteNode($N_{Expertise}$) 3) Rename(N_{New}, "New", "Add") 4) Modify(N_{New}, changes, "{Id,Name,Expertise}", "{Id,Name}") 5) deleteAllLink($N_{e?}$) 6) deleteNode($N_{e?}$) 7) Modify($N_{postNew}$, value, "{Id'=i?, Name'=n?, Expertise'=e?}", "{Id'=i?, Name'=n?}") 8) Rename($N_{Affiliate}$, "Affiliate", "Join")
Conflicts	C_1:{0,0, concurrent-renaming} C_2:{2,8, rename-deleted-element} C_3:{3,4, concurrent-update} C_4:{6,7, concurrent-update}

-deleted-element originating because the operation *Affiliate* has been removed in $delta_1$ while it has been renamed in $delta_2$. A resolution to this conflict could be to remove *Affiliate* (*i.e.* the renaming operation of $delta_2$ is dropped) as the task associated with it has been delegated to the operation *New* through the modifications made in $delta_1$. Clearly, this resolution requires user intervention. Another possible resolution to this conflict consists of dropping the delete operation of $delta_1$ that is causing the conflict. The third conflict is a *concurrent-update* originating because of concurrent modifications of the attribute *changes* of the operation *New*. Since the modified attribute is a set, it is possible to resolve this kind of conflicts by checking if one of the values is a subset of the other one, which is the case in this example. Dropping the update operation of $delta_2$ resolves this conflict. The last conflict is a *concurrent-update* originating because of concurrent modifications of the attribute *value* of the post-condition of the operation *New*. If a predicate is written as a conjunction of clauses, then it is possible to treat it as a set containing these clauses. Consequently, we could resolve this conflict by verifying if one of the values is a subset of the other, which is the case here. Dropping the update operation of $delta_2$ resolves this conflict.

Systematic conflicts resolutions based on operations' priorities could be applied. These priorities are chosen by users, which allow resolving a large number of conflicts. For example, in case of conflicts originating because of removals (e.g. *modify-deleted-element*, *modify-deleted-link, rename-deleted-element*, *move-deleted-element*,*link-without-source,* and *link-without-target*), a removal could be considered as an operation with less priority compared to an insertion, a modification or a renaming. Consequently, a resolution to all these conflicts is drop the delete operations causing them. The same heuristic could be used with other conflicts such as *rename-moved-element* and *modify-moved-element* if a moving operation is given a higher priority compared to a renaming or modification operation.

4.4 Merging

Merging is a direct process that consists of integrating (or applying) the changes made in the delta(s) after all conflicts have been discovered and resolved. Thus, merging is only possible when there is a resolution attached to every conflict discovered. These resolutions may require some user-interaction. Moreover, the order of operations in a delta is important to allow some changes to take place. For example, a rename operation involving an element *e* should always take place after any other modifications involving *e*. Merging consists of applying the operations contained in a unified and conflict-free delta to a specification. This delta is obtained through a process that involves combining the operations contained in two deltas while resolving the conflicts associated with them. **Figure 5** shows a possible result of merging the modifications made to the class *Professor* of **Figure 2**.

The above merge class integrates the modifications made after adopting specific conflict resolutions. The removal of the attribute *Expertise* does not satisfy the intension of the modifications made in one delta while the removal of the operation *Affiliate* does not satisfy the intension of the modifications made in the other delta. However, they have to be done to resolve the conflicts associated with these elements. Systematic conflict reso

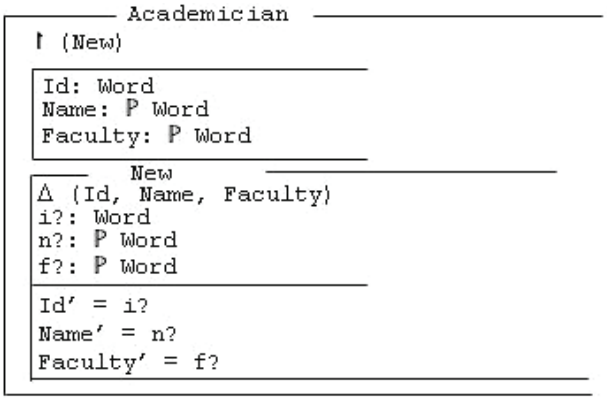

Figure 5. The result of merging two classes

lution saves time and effort and to some extend can preserve intensions. On the other hand, manual resolution provide a better platform for preserving intensions but they are time (and effort) consuming and are based on particular interpretations, which may lead to new inconsistencies in the merge results. Techniques to minimize conflicts could be used based on the idea of restricting the type of modifications a particular group of developers can make. Such as restricting modifications to addition only, creation of links only, etc. Thus, leading either to a reduction in the number of conflicts or to classes of conflicts that can be resolved easily and systematically.

5. Empirical Evaluation

A Java tool has been developed to evaluate and validate the proposed approach. It incorporates three major components. The first component processes a given OO formal specification and parses it into a graph. Currently, only Object-Z specifications are supported. The second component differentiates between two specifications represented as graphs after computing their similarities to produce a set of operations representing their delta. The third component integrates the edit operations contained in a unified delta to a base specification. This unified delta is obtained after combining the operations contained in two deltas according to the resolutions of the detected conflicts.

Merging is concerned about obtaining a consistent specification after the modifications contained in the delta(s) are integrated. Thus, it is critical to be able to detect and resolve as many conflicts as possible. Conflicts detection is dependent on the differencing algorithm used and the later is dependent on the accuracy of the similarity detection approach adopted. The proposed approach has been tested with three major case studies. They include a *university management system*, a *hotel management system* and an *online purchase system*. **Table 6** summarizes the details of the base version (V) of each specification as well as the modifications made to it (V_1 and V_2) and the actual number of conflicts arising because of the modifications made to the base specifications. Two different domain experts were in charge of modifying the base specifications according to requirements they think should be taken into consideration. The modifications made produced the specifications V_1 and V_2 respectively.

The combined base specifications contain a total number of 247 elements and links. The first versions of the specifications were obtained after performing 67 delta operations and the second versions were obtained through 82 delta operations made to the base specifications respectively. These modifications led to a total of 58 different conflicts.

The conflict detection and resolution approach was validated through the number of correct conflicts detected (positives), the number of all conflicts detected (positives and false positive) and the actual number of conflict that arise as a result of the modifications made (58 in the experiments made). Precision and recall metrics were used in the evaluation. Precision measures quality and is the ratio of the number of correct conflicts detected and resolved to the total number of conflicts detected. Recall measures coverage and is the ratio of the correct conflicts detected and resolved to the total number of correct conflicts. **Figure 6** shows the results obtained based on similarity threshold ranging from 0.5 to 0.9.

Perfect recall (100%) combined with a good precision (87%-98%) were obtained for thresholds raging from 0.5 to 0.7. Moreover, perfect recall (100%) combined with perfect precision (100%) were obtained for threshold ranging from 0.75 to 0.8. Furthermore, average to good recall (53%-86%) combined with perfect precision (100%) were obtained for thresholds ranging from 0.85 to 0.9. Out of 58 actual conflicts, only 8 conflicts were not detected for a high threshold of 0.85. These unidentified conflicts originate because of *too many* concurrent modifications and moving made to two classes namely *transactionInfo* and *shoppingCart* (and their elements), which led to many *concurrent-update* (4 cases), *concurrent-renaming* (2 case), *modify-moved-element* (1 case), and *concurrent-moving* (1 case) conflicts not being detected. For a threshold equals to 0.9, only 31 conflicts

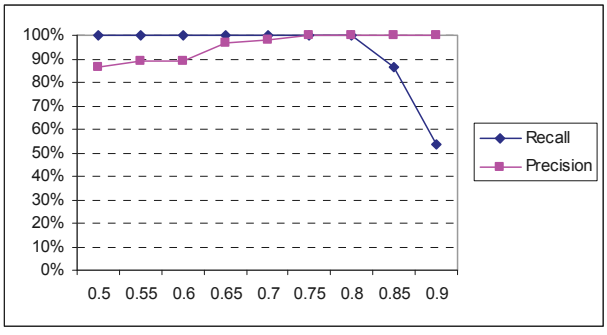

Figure 6. Experimental results

Table 6. Details of the experiments

	V		V_1			V_2		
	#Element	#Link	#Insertion	#Deletion	#Modification	#Insertion	#Deletion	#Modification
Case 1	28	27	13	0	3	15	2	7
Case 2	32	29	0	6	10	22	1	6
Case 3	71	60	4	22	9	4	22	3
Total	247		67			82		
#Conflicts	-				58			

were identified out of the actual 58 cases, which is indicated by the sharp drop of the recall. Consequently, if a high rate of positives is preferred while tolerating some negatives, a threshold value of 0.7 provides the best results.

The employed 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 because conflicts detection compares the operations contained in the deltas 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.

6. Related Work

In [13], an approach is proposed to detect the changes to XML documents. The proposed approach uses a delta that includes insertions, deletions and updates. Moving and renaming were not considered in the delta definition, thus, leading to the detection of conflicts that originate only because of deletions and updates. Moreover, the proposed approach is based on tree representation of the analyzed documents, which restricts its applicability to documents that can be represented as trees.

In [14], model merging was used to check the structural consistency of homogenous conceptual models described as graphs. The proposed approach constructs a merge model using given mapping information that equates the correspondences between the elements of the two graphs to be merged. Then, verifies it against some consistency constraints of interest. Consistency checking rules were described using the Relational Manipulation Language (RML). The consistency diagnostics obtained over the merge are projected back to the original models and their relationships. Lost update conflicts were not considered as structural conflicts were the main focus of the work. The presented work did not include experimental data on the rate of the discovered inconsistencies in terms of precision and recall.

In [15], structural and methodological model inconsistency is verified. The proposed approach does not support model merging but rather defines a model as a set of elementary construction operations, and consistency rules are defined and verified based on the type of operation involved and the effect an operation has on the model. Inconsistency rules were translated to Prolog queries and model construction operations to Prolog facts. The approach is tool supported were an XMI file containing a model is parsed into a sequence of model construction operations then a check engine is responsible of detecting elementary operations violating consistency rules within the sequence. The validation process used employed large UML models, and the results provided are mainly about time factors, *i.e.* efficiency and scalability.

In [16], 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 [17] 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 improving the accuracy of the change detection compared to the algorithm proposed in [18]. Finally, similar differencing algorithms were proposed in [19-21] to deal with different kind of software documents.

7. Conclusions and Future Work

An approach is proposed in this paper to detect and resolve conflicts that may occur when merging OO formal specifications. The differences between specifications are precisely calculated before a merge could take place. The difference is defined using primitive operations, acting on one element at a time, and containing traceability information enabling the reversal of their effects. The proposed approach deals with two major groups of conflicts: lost update and structural conflicts. Conflicts have been classified and formally defined as rules and a systematic approach is used to verify the calculated differences against these rules to detect any conflicts that may occur when integrating the changes made to a base specification. For every identified conflict a resolution is either derived systematically (pre-defined resolutions), or through user interaction (e.g. choosing among possible resolutions, etc). The experimental results obtained have validated the correctness and efficiency of the proposed approach as the majority of the conflicts contained in the studied specifications were systematically and cheaply (time and space) discovered. As an improvement to the proposed approach, optimizing the delta calculation and providing means to compress its content could be explored as it leads to a better efficiency. Finally, it is important to run more experiments using larger specifications.

REFERENCES

[1] P. Sriplakich, X. Blanc and M. P. Gervais, "Supporting Collaborative Development in an Open MDA Environment," *Proceedings of* 22*nd IEEE International Conference on Software Maintenance*, Los Alamitos, 2006, pp. 244-253.

[2] A. Boronat, J. A. Carsi, I. Ramos and P. Letelier, "Formal Model Merging Applied to Class Diagram Integration," *Electronic Notes in Theoretical Computer Science*, Vol. 166, No. 1, 2007, pp. 5-26.

[3] C. L. Ignat and M. C. Norrie, "Flexible Definition and Resolution of Conflicts through Multi-level Editing," *Proceedings of IEEE* 2*nd International Conference on Collaborative Computing*, Atlanta, 2006, pp. 10-19.

[4] G. Kontonya and I. Sommerville, "Requirements Engineering Process and Techniques," John Wiley, UK, 2002.

[5] M. G. Hinchey, "Industrial-Strength Formal Methods in Practice," Springer, 2008.

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

[7] A. Mehra, J. Grundy and J. A Hosking, "Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design," *Proceedings of ACM/IEEE International Conference on Automated Software Engineering*, Long Beach, 2005, pp 204-213.

[8] S. Fortsch and B. Westfechtel, "Differencing and Merging of Software Diagrams-State of the Art and Challenges," *Proceedings of International Conference on Software and Data Technologies*, Barcelona, 2007, pp. 90-99.

[9] K. Fogel and M. Bar, "Open Source Development with CVS," 3rd Edition, Paraglyph Press, 2003.

[10] D. E. Perry, H. P. Siy and L. G. Votta, "Parallel Changes in Large Scale Software Development: An Observational Case Study," *Proceedings of International Conference on Software Engineering*, Kyoto, 1998, pp. 251-260.

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

[12] F. Taibi, F. M. Abbou and M. D. Alam, "A Matching Approach for Object-Oriented Formal Specifications," *Journal of Object Technology*, Vol. 7, No. 8, 2008, pp. 139-153.

[13] S. Ronnau, C. Pauli and U. M. Borghoff, "Merging Changes in XML Documents Using Reliable Context Fingerprints," *Proceedings of* 8*th ACM symposium on Document Engineering*, Sao Paulo, 2008, pp. 52-61.

[14] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook and M. Chechik, "Consistency Checking of Conceptual Models Via Model Merging," *Proceedings of* 15*th IEEE International Requirements Engineering Conference*, New Delhi, 2007, pp. 221-230.

[15] X. Blanc, I. Mounier, A. Mougenot and T. Mens, "Detecting Model Inconsistency through Operation-Based Model Construction," *Proceedings of International Conference on Software Engineering*, Leipzig, 2008, pp. 511-519.

[16] Z. Xing and E. Stroulia, "Differencing logical UML models," *Journal of Automated Software Engineering*, Vol. 14, No. 2, 2007, pp.215-259.

[17] Y. Wang, "X-Diff: An Efficient Change Detection Algorithm for XML Documents," *Proceeding of* 19*th International Conference on Data Engineering*, Bangalore, 2003, pp. 519-530.

[18] A. Marian, "Detecting Changes in XML Documents," *Proceedings of* 18*th International Conference on Data Engineering*, San Jose, 2002, pp. 41-52.

[19] P. Apiwattanapong, N. Orso and M. J. Harrold, "A Differencing Algorithm for Object-Oriented Programs," *Proceedings of* 19*th International Conference on Automated Software Engineering*, Linz, 2007, pp. 2-13.

[20] U. Kelter, J. Wehren and J. Niere, "A Generic Difference Algorithm for UML Models," *Proceedings of Software Engineering Conference*, Brisbane, 2005, pp. 105-116.

[21] T. Oda and M. Saeki, "Generative Technique for Version Control Systems for Software Diagrams," *Proceedings of International Conference on Software Maintenance*, Budapest, 2005, pp. 515-524.

