Efficient Regression Testing of Ontology-Driven Systems

Mijung Kim†, Jake Cobb†, Mary Jean Harrold†, Tahsin Kurc*, Alessandro Orso†, Joel Saltz*, Andrew Post*, Kunal Malhotra†, and Shamkant B. Navathe†

†College of Computing
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.
{mijung.kim|jcobb|harrold|orso|kmalhotra7|sham}@cc.gatech.edu

*Center for Comprehensive Informatics
Emory University
Atlanta, Georgia, U.S.A.
{tkurc|jsaltz|arpost}@emory.edu

ABSTRACT

To manage and integrate information gathered from heterogeneous databases, an ontology is often used. Like all systems, ontology-driven systems evolve over time and must be regression tested to gain confidence in the behavior of the modified system. Because rerunning all existing tests can be extremely expensive, researchers have developed regression-test-selection (RTS) techniques that select a subset of the available tests that are affected by the changes, and use this subset to test the modified system. Existing RTS techniques have been shown to be effective, but they operate on the code and are unable to handle changes that involve ontologies. To address this limitation, we developed and present in this paper a novel RTS technique that targets ontology-driven systems. Our technique creates representations of the old and new ontologies, compares them to identify entities affected by the changes, and uses this information to select the subset of tests to rerun. We also describe in this paper OntoReTest, a tool that implements our technique and that we used to empirically evaluate our approach on two biomedical ontology-driven database systems. The results of our evaluation show that our technique is both efficient and effective in selecting tests to rerun and in reducing the overall time required to perform regression testing.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Algorithms, Reliability

Keywords

Regression testing; ontology-driven systems

1. INTRODUCTION

Scientific research in many areas, such as biomedicine, relies heavily on the use of complex database systems. These systems typically include a software layer that accesses multiple databases hosted at different sites, correlates the information gathered from the different databases, and performs various types of analyses on the resulting data. Often, the key to correlating the information from the different data sources is the use of an *ontology*: a formal description of a domain in terms of the concepts within that domain and their relationships to each other.

The existence of a unified ontology makes it easier to aggregate and integrate data in a given domain, even if they reside in different databases, to answer queries about such data, and to share information among users of the database system. For this reason, the use of ontologies, which was initially driven only by the need to interpret and consolidate data on the web, has quickly become common practice in the context of database applications (e.g., [19, 27]). Nowadays, ontologies are used to organize an increasing number of data domains, such as geoscience, biological science, finance, medical research, and healthcare delivery.

Like most software systems, ontology-driven database systems are tested to gain confidence in their behavior. However, unlike traditional software systems, tests for this kind of system consist of queries that are submitted to the database system and whose outcomes are checked against an expected result.[1] These databases and their underlying ontologies evolve over time because new data sources are integrated, mappings are changed, and ontology terms are added or deleted. Thus, these systems must be retested to gain confidence that they still behave as expected. This kind of testing, called *regression testing*, can be extremely expensive and can consume the majority of the overall testing and maintenance budget [5, 13, 18].

One way to perform regression testing is to rerun all available tests on the new version of the system, which guarantees that tests that may reveal problems in the new version are re-executed. Although effective, this strategy is clearly inefficient. Typically, only a subset of the existing tests is affected by the changes and should be rerun—rerunning unaffected tests results in unnecessary resource consumption and, ultimately, costs. To provide a concrete example, software engineers developing the Analytic Information Warehouse system [1] at Emory have reported to us that running all test queries for one of their database systems can take multiple days. For this reason, one of the key tasks in regression testing is *regression test selection (RTS)*—selecting a subset of existing tests to rerun on the modified software.

[1]In the rest of the paper, we use the terms test, test case, and test query interchangeably.

When a large number of tests are unrelated to the changes, and thus, irrelevant for the detection of faults caused by the changes, RTS can provide significant savings over running all existing tests.

Because of the importance of this problem, there is a large body of work on RTS, and researchers have presented many RTS techniques (e.g., [6, 11, 17, 26, 28, 31, 32]). All of these techniques are designed for traditional software and share a common characteristic: they focus on changes made to the application code and do not consider changes made to non-code components. In particular, these techniques would not work for ontology-driven database systems, because the changes in these systems are not in the code, but rather in the semantics and description of the data.

In this paper, we present a novel RTS technique that overcomes the limitations of existing RTS techniques in handling ontology-driven systems. The technique first constructs a graph-based representation of both the original (O) and the modified (O') ontologies. The technique then performs a comparison of the two graphs to identify three sets of entities in the ontologies: entities that were added to O', entities that were removed from O, and entities that are affected by the changes between O and O'. Finally, the technique uses the results of the comparison to identify the tests that need to be rerun on the new version of the system—those tests associated with the affected entities.

The main benefit of our technique is that, based on the changes in the ontology, it selects only those tests that need to be rerun. Thus, the approach can reduce significantly the time, the computational resources, and the cost of retesting a system. Another benefit of our technique is that it detects changes (additions and deletions) in an ontology, and computes the effects of the changes. This information can be provided to developers for understanding the impact of the changes, and for assisting in creating new tests, where needed. A third benefit of our technique is that it is scalable to large ontologies. Thus, unlike existing differencing techniques, our technique can provide developers with valuable information about differences and their effects for real-world ontologies.

In this paper, we also describe a prototype tool, called OntoReTest, that implements our technique, and that we used to assess the efficiency and effectiveness of our approach. We used OntoReTest to perform studies on several versions of two real subjects in the biomedical domain: a large informatics system for data warehousing of clinical data (i2b2) [20] and an ontology for genomic information provided by The Gene Ontology Consortium [3]. The results of our studies show that, for our subjects, our approach is effective and can reduce considerably the number of test queries that need to be rerun after a modification. Our results also show that the technique is efficient in that the time required for the analysis (i.e., the construction of the models, their comparison, and the identification of the affected tests) plus the time needed to rerun the selected tests is consistently and significantly less than the time needed to rerun all available tests. Overall, our results provide initial but clear evidence that our technique can reduce the overall cost of regression testing of ontology-driven systems.

This paper makes the following contributions:

- A clear identification, definition, and discussion of the problem of regression testing of ontology-driven database systems.
- A novel regression-test-selection technique for ontology-driven systems that accounts for changes in the underlying ontologies rather than in the code.
- A prototype tool, OntoReTest, that implements the technique, and studies, performed using OntoReTest on two systems in the biomedical domain, that show the effectiveness and efficiency of the technique.

2. ONTOLOGIES AND MOTIVATION

In this section, we present background on ontologies and illustrate them with an example. We also present a use-case scenario that motivates the need for our technique.

2.1 Ontologies

An *ontology* represents knowledge in a domain by presenting a common vocabulary that is shared in that domain. The knowledge in an ontology is represented as (1) a set of classes that specify the concepts about the domain and (2) relationships among those classes [7]. There are four main components of an ontology: classes, properties, restrictions, and individuals [22]. We describe each component, and illustrate it with an example of a pizza ontology,[2] represented as graph in Figure 1.

A *class* describes a concept in a domain. For our example, the nodes in Figure 1 represent classes in the pizza domain: `Pizza` and `Topping`. Each class can have subclasses. In Figure 1(a), class-subclass relationships are represented by solid directed edges between class nodes. For example, `VeggiePizza` is a subclass of `Pizza`.

A *property* describes a relationship between classes. (A property can also describe a relationship between *individuals*, as we discuss later in this section.) For example, `Pizza` and `Topping` in Figure 1(a) are linked by a property, `hasTopping`, represented as a dashed directed edge labeled with the name of the property.

A *restriction* is a constraint that a class must satisfy. In our example, restrictions are represented as dashed directed edges labeled with the name of the referenced property and with the constraint type in square brackets. For example, according to Figure 1(a), `Margherita` must satisfy two restrictions: `hasTopping[some] Tomato` and `hasTopping[some] Mozzarella`. These restrictions reference the `hasTopping` property, which links `Pizza` and `Topping`, and specify that `Margherita` must have the `hasTopping` property for at least one (*some*) instance of `Tomato` and at least one (*some*) instance of `Mozzarella`. That is, `Margherita` has at least one `Tomato` topping and at least one `Mozzarella` topping. Other constraint types are represented similarly.

An *individual* corresponds to an instance of a class. Individuals of a class have the same set of attributes (i.e., properties or restrictions) as the class to which the individuals belong. For example, a restaurant may sell a pizza named "Mushroom Deluxe," which is an individual of the class `MushroomPizza` in Figure 1(b). "Mushroom Deluxe" shares the attributes with `MushroomPizza`—it is an instance of `VeggiePizza` as well as `Pizza` and includes a restriction `hasTopping[some]` for an individual of `Mushroom`. Because individuals can change only if the class to which they belong changes, representing them in our graph would only add redundant information.

[2]The pizza ontology is a widely-used for ontology tutorials; it is publicly available at `www.co-ode.org/ontologies/pizza/`.

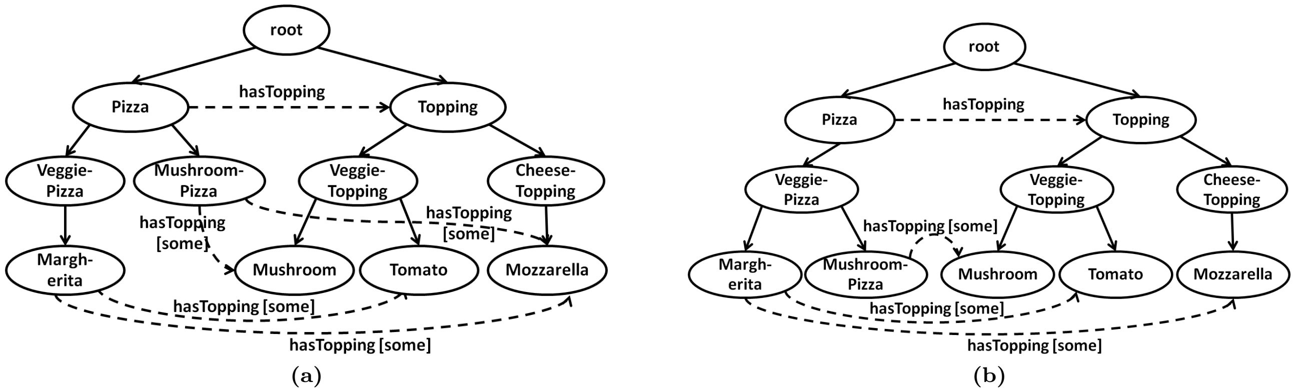


Figure 1: Example pizza ontology, represented as a graph G (a), and a changed version of the pizza ontology, represented as a graph G' (b).

Table 1: Example set of tests for the pizza ontology

Test		Entities
t_1	There are at least 3 veggie pizzas	`VeggiePizza`
t_2	Pizza must include mushroom pizza	`Pizza`, `MushroomPizza`
t_3	Mushroom pizza must have mozzarella toppings	`MushroomPizza hasTopping [some] Mozzarella`
t_4	There must be at least 1 veggie topping	`VeggieTopping`

So far, we have discussed the main components of an ontology. In this paper, we do not discuss other minor components, such as sub-properties, datatype properties, and cardinality restrictions, that our approach can handle but that would unnecessarily complicate the discussion.

Table 1 shows an intuitive description of four tests that may be used to test a system based on the pizza ontology, along with entities in the ontology with which they are associated. In the table, the first column (Test) lists the test number and its description, and the second column (Entities) shows the entities in the pizza ontology that are associated with the test. As discussed in Section 1, actual tests would consist of queries to the database together with expected query results. Test t_1, for instance, would consist of a query that counts the number of `VeggiePizza` instances in the database system, including the instances of the subclasses of `VeggiePizza`, such as the following:

```
SELECT COUNT(*) AS result FROM orders WHERE type
IN (SELECT term FROM terms WHERE term='VeggiePizza'
OR ancestor='VeggiePizza')
```

Test t_1 would check that the value of `result` is at least 3.

2.2 Motivating Example

Ontologies change over time, and when changes in an ontology occur, an ontology-driven system must be retested. Consider again the pizza ontology, represented as graph G in Figure 1(a). Consider also the changed version of this ontology, represented as graph G' in Figure 1(b). As the figure shows, there are two changes from G to G': (1) `MushroomPizza` is a subclass of `Pizza` in G and a subclass of `VeggiePizza` in G'; and (2) the `MushroomPizza hasTopping[some] Mozzarella` restriction does not appear in G'.

For the first change, because `MushroomPizza` has been moved, `VeggiePizza` now has two subclasses in G'. However, `Pizza` still has three subclasses: `VeggiePizza`, `Margherita`, and `MushroomPizza`. Moreover, `MushroomPizza` has no subclass in G or G'. Thus, for this change from G to G', only tests associated with `VeggiePizza` (i.e., t_1) may behave differently with the changed ontology because, although `MushroomPizza` has been moved, `MushroomPizza` and `Pizza` have the same subclasses in G and G'. Therefore, any tests associated with `MushroomPizza` (i.e., t_2) will return the same results if run on a database that uses the new ontology and on one that uses the original ontology. Therefore, t_2 does not need to be rerun.

For the second change, the deletion of the restriction `MushroomPizza hasTopping[some] Mozzarella` means that tests associated with this restriction (i.e., t_3) may behave differently and must be rerun. Finally, because there are no changes to `VeggieTopping` between G and G', tests associated with `VeggieTopping` (i.e., t_4) do not need to be rerun.

In summary, for the changes to the pizza ontology shown in Figure 1, only t_1 and t_3 must be rerun.

3. ALGORITHM

In this section, we present our algorithm for selecting tests to rerun based on a changed ontology.

3.1 Overview

Our algorithm, SELECTTESTS (Algorithm 1), inputs a graph, G, that represents the original ontology O, and a graph, G', that represents the changed ontology O'. An ontology graph[3] is a set of nodes N and a set of directed edges E. A *node* $n \in N$ represents a class. A *subclass edge* represents a subclass relationship, and is indicated as $e_s = (s,t) \in E$, where $s,t \in N$ are the source and target of the edge, respectively. A *property edge* represents either a property or a restriction, and is indicated as $e_p = (s,t,p) \in E$, where $s,t \in N$ are the source and target nodes, respectively, and p is the property name along with any restrictions. SELECTTESTS also inputs a matrix, M, that associates tests in T with entities in O. In general, matrix construction can be performed automatically by parsing tests and identifying

[3]Because there is no standard model for representing ontologies, for our work, we created a graph representation that supports explicit representation of ontologies.

```
Algorithm: SelectTests
Input :
  $G$, graph of the original ontology, $O$
  $G'$, graph of the changed ontology, $O'$
  $M$, matrix that maps $G \to T$
Output:
  ADD, set of added nodes and property edges in $G'$
  DELETE, set of deleted nodes and property edges in $G$
  $T'$, set of tests selected from $T$
Use :
  $PropOut(s)$, property edge triples $\langle s,t,p \rangle$ from node $s$
  $Succ(n)$, set of successor nodes of $n$
  $Descend_G(n)$ and $Descend_{G'}(n')$, set of descendants
     via subclass edges of node $n$ and $n'$ in $G$ and $G'$
  AFFECT, set of affected nodes and property edges in $G$
begin
  // Phase 1: Compute descendants of nodes
  foreach node $n \in G$ and $n' \in G'$ in reverse
        topological order on the subclass edges do
    $Descend_G(n) \leftarrow Succ(n) \bigcup_{s \in Succ(n)} Descend_G(s)$
    $Descend_{G'}(n') \leftarrow Succ(n') \bigcup_{s' \in Succ(n')} Descend_{G'}(s')$
  end
  // Phase 2: Identify changes,compute effects
  ADD $\leftarrow \{\langle n,-,- \rangle : n \in G'\}$ // Initialize ADD
  DELETE $\leftarrow \emptyset$ // Initialize DELETE
  AFFECT $\leftarrow \emptyset$ // Initialize AFFECT
  foreach node $n \in G$ do
    if $\exists n' \in G' : label(n) = label(n')$ then
      ADD $\leftarrow$ ADD $-\{\langle n',-,- \rangle\}$// remove $n'$
      if $Descend_G(n) \neq Descend_{G'}(n')$ then
        AFFECT $\leftarrow$ AFFECT $\cup \{\langle n,-,- \rangle\}$
      end
      if $PropOut(n) \neq PropOut(n')$ then
        DELETE $\leftarrow$ DELETE
            $\cup(PropOut(n) - PropOut(n'))$
        AFFECT $\leftarrow$ AFFECT
            $\cup(PropOut(n) - PropOut(n'))$
        ADD $\leftarrow$ ADD $\cup(PropOut(n') - PropOut(n))$
      end
    end
    else // when match of $n$ is NOT in $G'$
      DELETE $\leftarrow$DELETE $\cup\{\langle n,-,- \rangle\} \cup PropOut(n)$
      AFFECT $\leftarrow$AFFECT $\cup\{\langle n,-,- \rangle\} \cup PropOut(n)$
    end
  end
  // Find new property edges from nodes in ADD
  foreach $\langle n',-,- \rangle \in$ ADD do
    ADD $\leftarrow$ ADD $\cup PropOut(n')$
  end
  // Phase 3: Select tests from $T$
  $T' \leftarrow \emptyset$ // Initialize $T'$
  foreach $a \in$ AFFECT do
    $T' \leftarrow T' \cup$ tests associated with $a$ in $M$
  end
  return ADD, DELETE, $T'$;
end
```

Algorithm 1. SelectTests

the ontology entities in the tests. In Section 4, we describe how we constructed the matrix for our experiments.

SelectTests identifies changes from O to O' by comparing G and G'. The algorithm computes three sets that represent the changes in classes and properties: (1) ADD, the set of entities added to O'; (2) DELETE, the set of entities deleted from O; and (3) AFFECT, the set of entities affected by the changes from O to O'. To collect information about entities that are related to ontology changes, SelectTests represents nodes and edges as triples. A triple, $\langle n,-,- \rangle$, where n is the node and the second and third elements of the triple are empty, represents a node, which corresponds to a class in an ontology. A triple $\langle s,t,p \rangle$, where s is the source node of property edge p and t is the target node of property edge p, represents a property edge, which corresponds to a property or a restriction in an ontology.

As the example in Section 2.2 illustrates, a node can be affected directly or indirectly by changes, and the algorithm must select tests associated with both kinds of effects. If a node itself is deleted, there is a *direct* effect. If the set of subclasses of a node has changed because of the addition or deletion of other nodes, there is an *indirect* effect. A change in the set of subclasses is an indirect effect because it potentially changes the respective results of tests associated with the parent class. To find all data instances of a given class from an ontology, the data source must be searched for instances of any subclasses because they are, by definition, instances of the given class. After computing the changes, the algorithm uses AFFECT and M to select T', which contains those tests in T that need to be rerun. The algorithm then returns ADD, DELETE, and T'

For simplicity (as discussed in Section 2.1), in this paper we focus only on the main components of an ontology. Our algorithm can easily handle the ontology components that we do not discuss, such as cardinality restrictions, by representing them as additional nodes and edges in the graph.

SelectTests consists of three phases. In Phase 1, for each node n in G and G', SelectTests computes $Descend_G$(n) by collecting all descendants of n through subclass edges. These sets of descendants will be used in Phase 2 to compute entities affected by changes. In Phase 2, SelectTests identifies the changes between G and G', and computes the effects of the changes in nodes and property edges. In Phase 3, SelectTests simply selects tests associated with affected entities in G using the matrix M.

3.2 Algorithm Details

In this section, we present the details of SelectTests and illustrate them with an example.

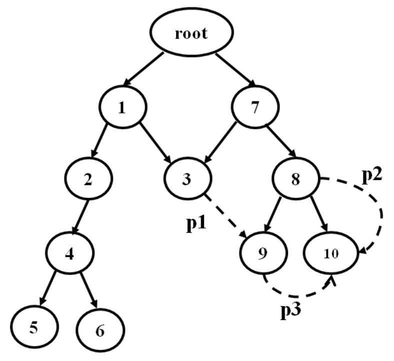
3.2.1 Phase 1: Compute Descendants

SelectTests computes descendants of each node n in G and G', $Descend_G$ and $Descend_{G'}$, respectively (lines 1-3). The algorithm performs this computation by traversing G and G' in a reverse topological order[4] on the subclass edges in the graphs beginning with the leaf nodes.

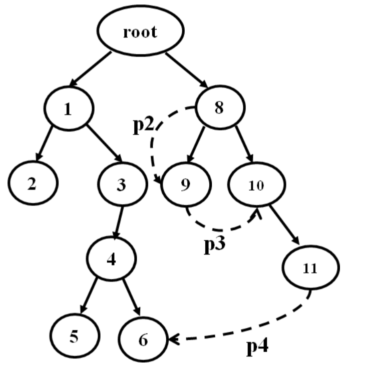
For each node n, $Descend(n)$ is the union of the successors of n and $Descend$ of all successors of n, where the successors of n are the target nodes of outgoing subclass edges from n. SelectTests requires only one iteration of this computation because the subgraph induced by subclass edges is acyclic.

To illustrate, consider Figure 2, which shows graphs G and G' that represent an original ontology and a changed version of that ontology, respectively. Suppose that SelectTests computes $Descend_G$ using the reverse topological order for G $(10, 9, 8, 3, 7, 6, 5, 4, 2, 1, root)$. In this case, $Descend_G(10)$ is empty because node 10 has no successors.

[4]A *reverse topological ordering* of a directed graph is a linear ordering of the graph's nodes such that for every pair of edges (i, j), j appears before i in the ordering.



(a)



(b)

Entity	Test Cases	Entity	Test Cases
1	t_1	8	t_9
2	t_8	9	t_{10}
3	t_1, t_2	10	t_{10}
4	t_7	11	t_{11}
5	t_5, t_6	p1	t_3
6	t_5	p2	t_{12}
7	t_{13}	p3	t_4, t_5

(c)

Figure 2: Graph G for ontology O (a), graph G' for a changed version of O (b), and test matrix M (c).

Table 2: Descendants of selected nodes in Figure 2(a) and (b)

n	$Descend_G(n)$	$Descend_{G'}(n)$
1	{2, 3, 4, 5, 6}	{2, 3, 4, 5, 6}
2	{4, 5, 6}	{}
3	{}	{4, 5, 6}
4	{5 ,6}	{5, 6}
5	{}	{}
6	{}	{}

Likewise, $Descend_G(9)$ is also empty. $Descend_G(8)$ is the union of node 8's successors ($\{9, 10\}$) and the descendants of node 8's successors ($\{\}$). As a result, $Descend_G(8) = \{9, 10\} \cup Descend_G(9) \cup Descend_G(10) = \{9, 10\}$. For another example, node 2 has only node 4 as a successor, but $Descend_G(4) = \{5, 6\}$. Thus, $Descend_G(2) = \{4, 5, 6\}$. Table 2 shows the descendants for nodes $(1, ..., 6)$ in G and nodes $(1, ..., 6)$ in G'.

3.2.2 Phase 2: Identify Changes,Compute Effects of Changes

SelectTests supports two types of local changes: addition and deletion. The algorithm computes a set of added entities (ADD), a set of deleted entities (DELETE), and a set of affected entities (AFFECT). Line 4 initializes ADD with the triples for each node in G'; the algorithm will subsequently remove those nodes that exist both in G and G', and the remaining nodes in ADD will represent those added to G'. Line 5 initializes DELETE with an empty set; the algorithm will add those nodes that exist in G but not in G'. Line 6 initializes AFFECT with an empty set; the algorithm will add those nodes that are affected by the changes. For the example shown in Figure 2, after lines 4-6 of the algorithm, ADD is $\{\langle root, -, -\rangle, \langle 1, -, -\rangle, ..., \langle 11, -, -\rangle\}$, whereas DELETE and AFFECT are both $\{\}$.

After initialization, SelectTests visits each node n in G (line 7) and attempts to find n's match by finding $n' \in G'$ whose label is the same as n's. If n' is found (line 8), the algorithm first removes $\langle n', -, -\rangle$ from ADD (line 9) so that it is not treated as an added node. Then, in line 10, the algorithm compares $Descend_G(n)$ and $Descend_{G'}(n')$, which were computed in Phase 1. If they differ, the algorithm adds $\langle n, -, -\rangle$ to AFFECT (line 11). In this case, although n and n' match with the same label, n is affected because the difference between $Descend_G(n)$ and $Descend_{G'}(n')$ reflects changes in the subclasses.

To illustrate, consider again Figure 2. For simplicity, consider only the left subtree of *root*, which includes nodes $(1, ..., 6)$. Because these nodes exist in both G and G', the matches are found (line 8) and the nodes are removed from ADD (line 9). Table 2 shows that only $Descend(2)$ and $Descend(3)$ differ even though the location of the subtree rooted at node 1 has changed. Despite this location change, because $Descend_G(4)$, $Descend(5)$, and $Descend(6)$ are still the same as those of $Descend_{G'}$, the algorithm does not consider nodes 4, 5, and 6 as affected classes because any tests associated with these classes will behave the same using O or O'. Likewise, because $Descend_G(1)$ is also the same as $Descend_{G'}(1)$, the algorithm does not include node 1 in AFFECT. Thus, the algorithm adds only $\langle 2, -, -\rangle$ and $\langle 3, -, -\rangle$ to AFFECT for the left subtree of *root* in G.

Line 12 of SelectTests inspects a set of outgoing property edges from n and n', $PropOut(n)$ and $PropOut(n')$, respectively, and compares them. If they differ, the algorithm adds the deleted property edges to DELETE and AFFECT (lines 13-14) and the added property edges to ADD (line 15). For the deleted property edges, the algorithm computes the difference in the outgoing property edges of n and n' ($PropOut(n) - PropOut(n')$), whose triples refer to the deleted property edges in G—the edges exist in G, but not in G'. Thus, the algorithm adds these triples to DELETE (line 13). In addition, because the algorithm builds AFFECT in terms of entities in G and the deleted property edges are changed entities in G, the algorithm adds the triples to AFFECT as well (line 14). For the added property edges, the algorithm computes the difference in the outgoing property edges of n' and n ($PropOut(n') - PropOut(n)$), whose triples refer to the added property edges in G'—the edges exist in G', but not in G. Thus, the algorithm adds these triples to ADD (line 15). In this case, however, because the algorithm builds ADD in terms of G', not G, it does not add these triples to AFFECT.

To illustrate, consider node 8 in Figure 2. Because nodes labeled "8" exist in both G and G', they are matched nodes. However, $PropOut(8) \neq PropOut(8')$ (line 12): $PropOut(8)$ in G is $\{\langle 8, 10, p2\rangle\}$ whereas $PropOut(8')$ in G' is $\{\langle 8, 9, p2\rangle\}$. Thus, lines 13-15 are invoked. In lines 13-14, the algorithm adds $PropOut(8) - PropOut(8')$ ($\{\langle 8, 10, p2\rangle\}$) to both AFFECT and DELETE. Likewise, in line 15, the algorithm adds $PropOut(8') - PropOut(8)$ ($\{\langle 8, 9, p2\rangle\}$) to ADD.

Next SelectTests handles the case in which a match for n (i.e., n') is not found (line 16). When the match is not found, n exists in G but not in G' because n has been deleted from G. For deleted nodes, the algorithm not only adds $\langle n, -, -\rangle$ to DELETE and AFFECT, but also inspects the set of outgoing property edges from n, $PropOut(n)$, and adds this set to DELETE and AFFECT (lines 17-18). For the example in Figure 2, because node 7 exists in G but not in G', when lines 17-18 of the algorithm are invoked, the algorithm adds $\langle 7, -, -\rangle$ to DELETE and AFFECT. However, because $PropOut(7)$ is empty, nothing is added in terms of the outgoing property edges from 7.

After the algorithm removes all triples for $n' \in G'$ that have a match $n \in G$ (line 9), ADD contains only triples for new nodes added to G'. To determine whether these nodes include any outgoing property edges (which are necessarily also newly added), the algorithm visits each added node n in ADD (line 19) and adds $PropOut(n)$ to ADD (line 20). For example, when the algorithm reaches line 19, ADD contains one node, node 11 in Figure 2(b), which also includes a new outgoing property edge $p4$. Line 20 of the algorithm inspects $PropOut(11)$, $(\{\langle 11, 6, p4\rangle\})$ and adds it to ADD (line 20).

After the algorithm completes the computation, the final states for the example in Figure 2 are

ADD=$\{\langle 11, -, -\rangle$, $\langle 8, 9, p2\rangle$, $\langle 11, 6, p4\rangle\}$,
DELETE=$\{\langle 7, -, -\rangle$, $\langle 3, 9, p1\rangle$, $\langle 8, 10, p2\rangle\}$, and
AFFECT=$\{\langle root, -, -\rangle$, $\langle 2, -, -\rangle$, $\langle 3, -, -\rangle$, $\langle 7, -, -\rangle$, $\langle 8, -, -\rangle$, $\langle 10, -, -\rangle$, $\langle 3, 9, p1\rangle$, $\langle 8, 10, p2\rangle\}$.

3.2.3 Phase 3: Select Tests

SelectTests selects all tests associated with entities in AFFECT (lines 21-23) to include in T'. Figure 2(c) shows the test matrix that associates entities in G with tests in T for the example ontology O. For the example, using the AFFECT set shown above, the algorithm adds $\{t_1, t_2, t_3, t_8, t_9, t_{10}, t_{12}, t_{13}\}$ to T'. The remaining tests in T that do not need to be run are $\{t_4, t_5, t_6, t_7, t_{11}\}$.

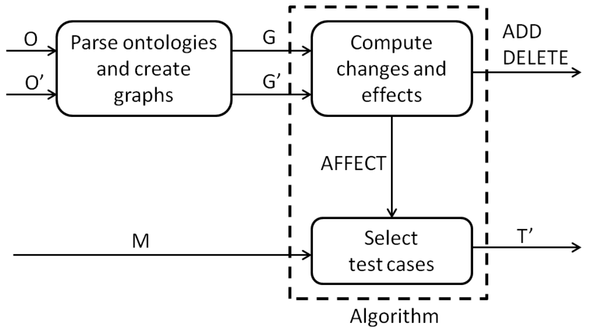
Finally, SelectTests returns ADD, DELETE, and T' (line 24). The runtime of SelectTests is linear in the size of G and G'; see [14] for the full complexity analysis.

3.2.4 Safety of SelectTests

An important aspect of an RTS technique is its safety. A *safe* RTS technique guarantees that the set of tests it selects to be rerun contains all tests in T that may behave differently when run on the changed system [28].

An ontology-driven system includes a database that locates a set of data instances associated with entities of an ontology. If a test (test query) references one or more classes, it will query against the set of data instances associated with each of those classes and their subclasses. Thus, tests will behave differently when they reference classes whose set of subclasses differ because of changes in the ontology. For example, when the subclasses of a class in the ontology change, the database will change to include instances of any new subclasses and exclude instances of any removed subclasses. SelectTests computes the transitive set of subclasses for each class in the original and modified ontologies, and classifies those classes that have a different set of subclasses as affected entities. Thus, it selects all tests associated with classes in which the set of data instances associated with those classes and their subclasses may have changed.

Additionally, if a test references a property or a restriction, it will query against the set of data instances associated with that property or restriction only. Tests will thus



O
O'
Parse ontologies and create graphs
G
G'
Compute changes and effects
ADD
DELETE
AFFECT
M
Select test cases
T'
Algorithm

Figure 3: Three components of OntoReTest.

behave differently when they reference deleted properties. For example, if the constraint type of the restriction on a property edge changes, the interpretation of that edge is potentially different. Because our graph representation includes restrictions in the property edge label, SelectTests detects changed restrictions as deletions of the original edges and additions of the new ones, and classifies the deleted original edges as affected entities. Thus, it will select all tests associated with the deleted restriction edge. Selection with respect to property changes will be performed in a similar manner because properties and restrictions are represented by the same type of edges.

3.2.5 Limitation of SelectTests

The main limitation of our algorithm is that knowledge not explicitly defined in an ontology (i.e., implicit concepts) will not be represented in our graph. To address this limitation, we run a reasoner engine on the ontology being modeled before constructing its graph model. This is a commonly-used approach in ontology languages because reasoner engines can deduce implicit knowledge. Such engines can, for instance, infer class-subclass relationships and determine whether a class is consistent (i.e., it can have instances).

4. EMPIRICAL EVALUATION

To evaluate the effectiveness and efficiency of algorithm SelectTests, we implemented it in a tool, OntoReTest, and used the tool to conduct studies on two large real-world ontologies. In this section, we describe the experimental setup, discuss the studies, and present the results.

4.1 Experimental Setup

4.1.1 Implementation

As Figure 3 shows, OntoReTest consists of components for (1) parsing the old and new ontologies and creating graphs, (2) computing changes and effects of the changes (Phases 1 and 2 of SelectTests), and (3) selecting tests associated with the effects of the changes (Phase 3 of SelectTests).

The first component inputs the old and new versions of ontology files, O and O', respectively, parses them, and creates graphs. The component uses several libraries to assist with parsing and reasoning about ontologies and to represent and traverse the graphs. First, the component parses O and O' using the OWL API,[5] and translates them into OWL (the Web Ontology Language) ontologies [2]. OWL is

[5]`http://owlapi.sourceforge.net/`

Table 3: Subjects used in the empirical studies

Subject	Classes	Properties & Restrictions	# Vers.	Test Cases	Coverage
`i2b2`	104967	0	7	1331	99.99%
`GO`	34636	11101	7	3499	52.75%

a W3C (World Wide Web Consortium) standard. The OWL API supports various input formats, such as RDF/XML [4], OWL/XML [12], and OBO [29], parses these formats, and translates them into OWL ontologies. Supporting multiple input formats lets OntoReTest be applicable to a variety of ontologies. Second, after parsing the ontologies, the component invokes the HermitT[6] OWL reasoner to compute inferable subclass relationships in each ontology. Third, the component constructs and outputs graphs G and G'. The component uses the JGraphT library[7] to represent G and G', with custom extensions for handling specific edge types.

Using G and G', the second component computes ADD, DELETE, and AFFECT. This component outputs ADD and DELETE, which will be reported to the user, and AFFECT, which will be used by the third component.

Finally, the third component inputs AFFECT along with M, which associates tests in the original test suite T with entities in G. This component outputs T', which are those tests in T associated with elements in AFFECT, as indicated by M. For our studies, we obtained M automatically for each subject by parsing the test queries and extracting the ontology terms contained in each test query.

4.1.2 Subjects

Our first subject is `i2b2` (Informatics for Integrating Biology and the Bedside) [20], which is a large informatics framework and system that targets data warehouses for clinical data in the biomedical domain. `i2b2` organizes data around terms (i.e., classes) from an ontology; all user queries in `i2b2` use terms from the ontology and do not query the backend database directly. `i2b2` clients interact with the `i2b2` server by exchanging XML-based messages over a REST (REpresentational State Transfer) interface [23], which is an architectural style of creating client-server web services. Table 3 provides information about the `i2b2` ontology.

To evaluate our approach against `i2b2`, we used real ontology changes from the Analytic Information Warehouse (AIW) project [1] of the Center for Comprehensive Informatics (CCI) at Emory University. Researchers in the AIW project—several of whom are collaborators on this work—used the standard `i2b2` ontology and customized it for the architecture of their system by adding, removing, and changing ontology classes. Although the intermediate versions used in the AIW project were not preserved, we had access to the original and current versions of the ontology, v_0 and v_1. This allowed us to simulate the incremental evolution of the ontology by creating five intermediate versions, $v_a, ..., v_e$, each of which contained a subset of the changes between v_0 and v_1. We were therefore able to use seven versions of the `i2b2` ontology in our experiments: v_0, v_a,..., v_e, and v_1. (Note that, because `i2b2` uses its own representation for ontologies, we converted each ontology into an OWL representation for use by SelectTests.)

We could not run the test queries from the AIW project because they involve sensitive health information that is protected by law and institutional policy. However, we were able to acquire test queries by using the actual user queries run against the public `i2b2` instance hosted at i2b2.org. These queries use classes from the standard `i2b2` ontology, which is the base ontology we used for evaluating the effects of changes. We initially gathered 3162 test queries this way. We eliminated redundant queries that used the same ontology terms and logical operators, which resulted in 1331 unique test queries that we used as our test suite for the studies. We also measured the coverage achieved by this test suite in terms of the ontology classes that it exercises, directly and indirectly,[8] and found that it covers 99.99% of the `i2b2` ontology entities, as reported in Table 3. Our collaborators within the AIW project confirmed that these public test queries are structurally similar to those used during the development of the AIW project.

Our second subject is the Gene Ontology (`GO`), which is an ontology for genomic information provided by The Gene Ontology Consortium [3]. `GO` is available in OBO format, which is compatible with OWL. Furthermore, `GO` is available as a relational database (e.g., MySQL) consisting of two parts: the *term database*, which contains the ontology terms and relationships, and the *annotated data set*, which contains instance data that is annotated with the `GO` terms. Table 3 provides information about `GO`. Although `GO` is smaller than the `i2b2` ontology, it includes properties and restrictions that the `i2b2` ontology lacks. (Note that all properties and restrictions in `GO` can be represented in our current representation.)

To evaluate our approach against `GO`, we used seven versions of `GO`, available publicly at the NCBO (National Center for Biomedical Ontology) BioPortal website.[9] `GO` is updated daily in the BioPortal. We used version 1.1.2132 as our base ontology, v_0, and selected five versions that were updated at various intervals after v_0 (from 1 day to 1.5 months): v_a to v_e. Finally, we chose a version with a large number of changes by selecting a version that was one-year old with respect to v_0. We labeled that version v_{-1}.

We used the relational database (MySQL) form of `GO`, which includes an annotated data set, as the target system for our tests. We gathered test queries for `GO` using example queries from the Gene Ontology Consortium's public wiki page.[10] More precisely, we selected seven queries and used them as query templates: two templates query the term database only, whereas the remaining templates query both the term database and the annotated data set. For each template, we generated 500 queries by selecting distinct `GO` terms at random from the term database, and substituting each selected term for the original term in the query template. One of the 3,500 queries we generated repeatedly failed to execute because of an out-of-memory error, so we used 3,499 unique queries as our tests. Also in this case, we measured the coverage achieved by the test suite on the ontology under test, and found that it covers 52.75% of the `GO` ontology entities.

[6]`http://hermit-reasoner.com/`

[7]`http://http://www.jgrapht.org/`

[8]An entity is *directly covered* when the entity is exercised because it is referenced in a test query and *indirectly covered* when the entity is exercised because it is a subclass of a directly-covered entity.

[9] `http://bioportal.bioontology.org/ontologies/1070`

[10] `http://wiki.geneontology.org/index.php/Example_Queries`

Table 4: Results of Study 1 for `i2b2`

Version Pair	ADD	DEL-ETE	AFF-ECT	Our approach / Retest all	% of tests selected
(v_0,v_a)	17982	15943	15945	626 / 1331	47.03
(v_0,v_b)	15887	1551	1553	150 / 1331	11.27
(v_0,v_c)	35347	9613	9615	136 / 1331	10.22
(v_0,v_d)	4584	4410	4412	78 / 1331	5.86
(v_0,v_e)	880	61	64	43 / 1331	3.23
(v_0,v_1)	74681	31578	31581	906 / 1331	68.07

Table 5: Results of Study 1 for `GO`

Version Pair	ADD	DEL-ETE	AFF-ECT	Our approach / Retest all	% of tests selected
(v_0,v_a)	26	0	57	5 / 3499	0.14
(v_0,v_b)	213	0	259	27 / 3499	0.77
(v_0,v_c)	350	16	523	56 / 3499	1.60
(v_0,v_d)	388	17	587	59 / 3499	1.69
(v_0,v_e)	534	25	835	74 / 3499	2.11
(v_0,v_{-1})	265	4181	7214	624 / 3499	17.83

4.2 Study 1: Effectiveness of Test Selection

The goal of this study is to evaluate the effectiveness of our approach in selecting tests for changes in ontologies. To do this, we

1. Ran OntoReTest on each pair of versions for each subject.
   - Recorded the sizes of ADD, DELETE, and AFFECT.
   - Recorded the number of tests selected from the original test suite.
2. Computed the percentage of tests selected from the original test suite.

Tables 4 and 5 present the results of the study for `i2b2` and `GO`, respectively. The first column of each table shows the version pair for the old and new versions of the ontology. The next three columns give the number of entities included in the result sets (i.e., ADD, DELETE, and AFFECT) produced by our algorithm. The next column shows the ratio of the number of tests selected by our approach to the number of tests in the test suite, and the last column shows the percentage of tests selected. For example, in Table 4, for (v_0,v_e), there are 880 added entities (ADD), 61 deleted entities (DELETE), 64 affected entities (AFFECT), and 43 tests selected out of 1331 in the test suite or 3.23% of the tests.

The data in Table 4 indicate that the percentage of tests selected for `i2b2` ranges from 3.23% to 68.07%. For (v_0,v_a), which includes the greatest number of changes among the intermediate versions, the percentage of selected tests is 47.03%, which reduces by more than half the number of tests to rerun on v_a. For (v_0,v_b) to (v_0,v_e), the percentage of selected tests is less than 12%, which illustrates the significant reduction in the number of tests to rerun that can be achieved by our algorithm. Furthermore, for (v_0,v_1), where a large number of changes are made (approximately 30,000 out of the 100,000 classes are changed), our approach selects less than 70% of the tests. Thus, even in the worst case for `i2b2`, our approach is effective in reducing by over 30% the number of tests that need to be rerun.

The data in Table 5 show that even better results are achieved for `GO` than for `i2b2`: the percentage of tests selected ranges from 0.14% to 17.83%. Because (v_0,v_a) and (v_0,v_b) include no entities in DELETE and the size of AFFECT is small, OntoReTest selects less than 1% of the tests. Because there are more changes in (v_0,v_c), (v_0,v_d), and (v_0,v_e), the tool selects more tests than it did for the first two version pairs, but the percentage is still only approximately 2%. For (v_0,v_{-1}), which is the comparison of the base and the year-old ontologies, the tool selects 17.83% of the tests. Thus, even in the worst case for `GO`, our approach reduces the number of tests that need to be rerun by over 80%.

Overall, the data in Tables 4 and 5 show that our approach is effective in selecting tests for two real-world ontologies. The results illustrate that our approach can achieve significant reduction in the number of tests that need to be rerun based on a changed ontology.

4.3 Study 2: Efficiency of Test Selection

The goal of this study is to evaluate the efficiency of our approach in selecting tests. To be efficient, the time required to select the tests plus the time required to rerun the selected tests must be less than the time required to rerun all the tests. To do this, we

1. Collected the running time of each test and calculated the sum of these running times, which gave the time required to rerun all tests (the Retest-all approach).
2. Ran each test five times and collected the minimum and maximum from the five runs.
3. Computed the *total time for regression testing*, which is the sum of the time that OntoReTest requires to select tests and time that it takes to run the tests that the tool selected (Section 4.2).
4. Compared the total time for regression testing with the Retest-all approach.

So that we could observe possible differences in running time due to network congestion, server load, etc. for `i2b2`, we performed this study not only on a public `i2b2` instance hosted by i2b2.org but also on a local instance in our local network. We performed the study for the `GO` database on an instance located in our local network.

Table 6 presents the results of the study. The first column shows the subjects. The second column shows Retest all and the version pairs, which are the same as those we used in Study 1. The next two columns list the minimum (MIN) and the maximum (MAX) running times (in seconds)[11] of tests for Retest all and for the tests selected for each version pair. Finally, the last column lists the running time of OntoReTest in seconds. For example, for Retest all for public `i2b2`, the MIN and MAX values of running time for all tests are 7528 seconds and 12104 seconds, respectively. The running time of OntoReTest is not applicable for Retest all because our analysis is not performed when all tests are rerun. For another example, for version pair (v_0,v_b) for local `i2b2`, the MIN and MAX running times for the selected tests are 782 seconds and 1034 seconds, respectively, and the running time for OntoReTest on (v_0,v_b) is 41 seconds.

Figure 4 presents the data from Table 6 as two graphs that show the MIN (in the upper graph) and MAX (in the lower graph) running time for our regression-testing approach as a percentage of the Retest-all approach. In each graph, the horizontal axis represents Retest all and the version pairs for three subjects: public `i2b2`, local `i2b2`, and `GO`. The vertical axes represent the total time for regression

[11]Rounded to seconds for all but (v_0,v_a) of `GO`.

Table 6: Results for Study 2

Subject	Version Pair	MIN (Seconds)	MAX (Seconds)	ONTO-RETEST (Seconds)
public `i2b2`	Retest all	7528	12104	n/a
	(v_0,v_a)	3803	6018	43
	(v_0,v_b)	834	1355	41
	(v_0,v_c)	822	1311	45
	(v_0,v_d)	441	716	40
	(v_0,v_e)	255	396	42
	(v_0,v_1)	5231	8386	40
local `i2b2`	Retest all	7107	9810	n/a
	(v_0,v_a)	3588	4859	43
	(v_0,v_b)	782	1034	41
	(v_0,v_c)	781	1139	45
	(v_0,v_d)	407	540	40
	(v_0,v_e)	234	314	42
	(v_0,v_1)	4924	6754	40
`GO`	Retest all	14220	20711	n/a
	(v_0,v_a)	0.012	0.284	96
	(v_0,v_b)	948	1617	89
	(v_0,v_c)	1014	2626	101
	(v_0,v_d)	1015	2629	91
	(v_0,v_e)	1385	3942	87
	(v_0,v_{-1})	9451	13280	95

testing (i.e., the time required to select tests plus the time to run the selected tests) as a percentage of the time required for the Retest-all approach. Because the total time for regression testing is calculated as a percentage of Retest all, in each graph, the height of the bars for Retest all is 100%.

Although running times for individual tests are not shown in Table 6, our results show that there was a high variance in the running times of individual tests in both subjects. For public `i2b2`, the running time of the tests varies from 2.6 seconds to 30.5 seconds, and the average running time of a test is 7.2 seconds. For local `i2b2`, the running time of each test ranges from 2.5 seconds to 52.3 seconds and the average running time of a test is 6.3 seconds. Table 6 shows that the running time of ONTORETEST on `i2b2` varies from 40 seconds to 45 seconds, which is a small portion of the entire running time of the selected tests. For all version pairs in the table, both MIN and MAX of the running times taken for public `i2b2` are greater than those for local `i2b2`. In particular, the MAX value of running all tests for public `i2b2` is 12104 seconds ($\simeq$ 3.4 hours) whereas the MAX value for local `i2b2` is 9810 seconds ($\simeq$ 2.7 hours). Thus, the data indicate the slowdown that may occur when regression testing is performed on a public server instance. However, as shown in Figure 4, the overall percentage of running time is similar for public `i2b2` and local `i2b2`. Moreover, the heights of the MIN and MAX bars for every version pair in public `i2b2` and local `i2b2` are also similar. For every version pair in the figure, the data illustrate the significant reduction in the time required to perform the regression testing: for (v_0,v_a) and (v_0,v_1), approximately 50% and 70%, respectively; for (v_0,v_b) and (v_0,v_c), approximately 11%; and for (v_0,v_d) and (v_0,v_e), approximately 5%.

For `GO`, the running time of each test ranges from 0.0006 to 5942.2 seconds and is 4.5 seconds on average. Additionally, as shown by the MIN and MAX values in Table 6, `GO` exhibits a significant variation between the MIN and MAX running times: tests that contain a number of JOIN clauses execute much longer than those without JOINs. Also, because we ran each test five times consecutively, the running

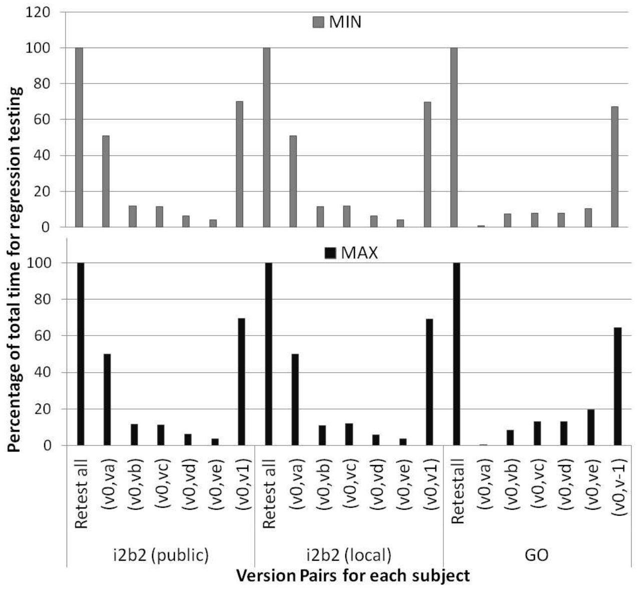


Figure 4: Graphical view of the results for Study 2.

times of the last four subsequent runs take relatively less time than that of the first run due to a caching effect; this leads to high variance between the MIN and MAX values for `GO` tests. This variation can be observed in Figure 4 as well. The figure shows that for (v_0,v_c), (v_0,v_d), and (v_0,v_e), the difference between the MIN and MAX values of the percentage is greater than that of other version pairs. However, despite the large variance, the percentage of running time required for regression testing is less than 20% for every version pair in `GO` except (v_0,v_{-1}), which contains the greatest number of affected entities (AFFECT).

Table 6 and Figure 4 illustrate the efficiency of our approach in selecting tests: they show a remarkable reduction in the running time for regression testing for every version pair compared to the Retest-all approach for all subjects. Moreover, for local `i2b2` and `GO` in Table 6, even if we compare the MIN running time of Retest all (i.e., 7107 seconds for local `i2b2` and 14220 seconds for `GO`, respectively) with the MAX running time of every other version pair in these two subjects, the MAX running time of regression testing by ONTORETEST is significantly less than the MIN running time of Retest all. We see this result in every version pair in public `i2b2` except (v_0,v_1). Therefore, our approach is efficient in test selection, and reduces the running time of regression testing significantly.

4.4 Threats to Validity

We consider several possible threats to the validity of our studies. Threats to internal validity concern possible errors in our implementation. To mitigate these threats, we performed unit tests when developing ONTORETEST using JUnit with different sizes of examples. For large ontology examples, we spot checked changes and verified that the results are correct. We also validated the scripts that we developed for selecting tests by comparing the results in the AFFECT set with the entities used in the test suites.

Threats to external validity arise when the results of the experiment cannot be generalized. We evaluated our technique with only two subject ontologies that are used in the bioinformatics domain. Thus, the performance of our tool may vary for other subjects in the bioinformatics domain or subjects in other domains. However, both of the systems and the versions we used are real, large systems and ontologies in common use. The test queries used as our tests in the `i2b2` study are those actually used in the public web client, as we discuss in Section 4.1.2. Furthermore, the developers of the AIW project [1] that are collaborating with us on this work confirmed that our tests are similar in terms of structure and coverage to the test queries they actually use for development within the project. The ones used for subject `GO` are derived from templates that correspond to real test queries available in the public domain.

Therefore, although more experimentation will help further validate our results, we believe that our results are likely to generalize to other systems and tests.

5. RELATED WORK

Little research has been performed on testing (and regression testing) of ontology-driven systems. However, some of the research on detecting changes in ontologies and on regression testing of databases is related to our work.

There have been several ontology-comparison techniques developed in the web services, artificial intelligence, and knowledge engineering domains. Noy and colleagues [24, 25] created the PromptDiff algorithm that finds various types of changes (e.g., add, delete, move, and rename) using a number of heuristic matchers they developed. Tury and Bieliková [30] also used heuristics in their tool OntoDiff for searching for equivalent (identical or similar) elements in ontology versions. However, because these techniques depend on heuristics, they cannot guarantee that they will detect all changes between two ontologies (i.e., they are not safe). In contrast, our algorithm, SELECTTESTS, does not depend on heuristics and is safe in selecting tests. Furthermore, our algorithm computes the effects of changes, in addition to detecting additions and deletions.

Klein and colleagues [15, 16] presented OntoView for ontology versioning and change detection. OntoView compares versions of RDF-based ontologies and computes the differences between classes and properties. However, OntoView supports only the RDF format, whereas ONTORETEST is compatible with various formats. In addition, because OntoView also partially depends on heuristics, it is not safe.

We tried to use these existing approaches to compute ontology differences. Unfortunately, the way they compare ontologies is more expensive, despite their use of heuristics, because they compute extra detailed information—information that is useless for our technique. In fact, these techniques do not scale and could not run on either of the ontologies we considered because of the size of the ontologies. Finally, we also found that these techniques may miss some changes, which would cause our technique not to select some potentially affected tests.

There are several existing techniques that perform regression test selection for databases. Willmor and Embury [33] presented a regression-testing technique for database-driven applications that safely performs test selection. Their technique determines whether changes to a program affect database states to select those tests that are associated not only with the program changes but also with the database states affected by the changes. There has also been research that addresses changes to database entities for selecting tests [10, 21]. The techniques select those tests that cover the database commands that associate with changed database entities. However, unlike these techniques, our technique addresses changes to ontologies and not database states. Moreover, our empirical studies demonstrate both the effectiveness and the efficiency of selecting tests for database systems based on changed ontologies.

Haftmann and colleagues [9] introduced an efficient regression-test-selection technique for database applications. They investigated executing tests in parallel and reducing database resets for database applications with improved running time of tests and reduced resources. Another technique of Haftmann and colleagues [8] is an efficient regression-testing technique that controls the order of test execution to address the limitation of the existing test tools not designed for database applications. Our work is related to theirs in that the goal of the research is to reduce the time required for regression testing. However, the techniques are complementary: our technique assists with efficient regression testing through effective test selection without controlling test executions (i.e., parallelized and ordered execution of tests), but the efficiency of our technique could be improved by integrating it with theirs.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a new technique for selecting tests based on changes in an ontology. Our technique uses a novel algorithm that compares graph representations of an ontology O and a changed version of that ontology O' to identify changes and effects of those changes. Using the effects of the changes, our technique selects tests to rerun that are associated with the affected entities in O.

In the paper, we also described an implementation of our technique, ONTORETEST, and studies on two real subjects to evaluate the efficiency and effectiveness of the technique. Our studies show that, for our subjects, ONTORETEST is effective in reducing the number of tests that need to be rerun based on changes to an ontology. The studies also show that, for our subjects, the technique is efficient and can provide significant reduction in regression-testing time.

There are several possible directions for future work. Our experiments show the reduction in regression testing that can be achieved by our technique. However, we performed our experiments on only two subjects, and in one particular domain—bioinformatics. In the future, we plan to evaluate our technique on other bioinformatics subjects and on subjects in other domains.

The technique we presented addresses regression test selection, which is an important regression-testing activity. There are other activities, such as test prioritization, test-suite augmentation, obsolete-test identification, and test repair that can be addressed. We believe that we can use the ADD, DELETE, and AFFECT sets computed by our algorithm to address some of these tasks. Our future work will include investigation of these additional regression-testing activities.

7. ACKNOWLEDGMENTS

This research was supported in part by grants NSF CCF-0541048, NSF CCF-0725202, NSF CCF-0964647, NSF CCF-1116210, NHLBI R24HL085343, NIH RC4MD005964, NCI HHSN261200800001E, NLM R01LM009239, NIH/CTSA PHS UL1RR025008, and NLM R01LM011119, and by an IBM

Faculty Award, an IBM Software Quality Innovation Award, and a Microsoft Research Software Engineering Innovation Foundation Award.

8. REFERENCES

[1] Analytic Information Warehouse. http://cci.emory.edu/cms/projects/aiw.html, 2012. [Online; accessed Feb-2012].

[2] G. Antoniou and F. V. Harmelen. Web Ontology Language: OWL. In *Handbook on Ontologies in Information Systems*, pages 67–92. Springer, 2003.

[3] M. Ashburner. Gene ontology: Tool for the unification of biology. *Nature Genetics*, 25:25–29, 2000.

[4] D. Beckett and B. McBride. RDF/XML Syntax Specification. Technical report, W3C, 2004.

[5] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[6] Y. F. Chen, D. S. Rosenblum, and K. P. Vo. TestTube: A system for selective regression testing. In *Proc. of ICSE '94*, pages 211–222, May 1994.

[7] T. R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5:199–220, June 1993.

[8] F. Haftmann, D. Kossmann, and A. Kreutz. Efficient regression tests for database applications. In *Proc. of CIDR '05*, pages 95–106, 2005.

[9] F. Haftmann, D. Kossmann, and E. Lo. A framework for efficient regression tests on database applications. *The VLDB Journal*, 16:145–164, January 2007.

[10] R. A. Haraty, N. Mansour, and B. Daou. Regression testing of database applications. In *Proc. of SAC '01*, pages 285–289, New York, NY, USA, 2001. ACM.

[11] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proc. of OOPSLA '01*, pages 312–326, Oct. 2001.

[12] M. Hori, J. Euzenat, and P. F. Patel-Schneider. OWL Web Ontology Language XML Presentation Syntax. http://www.w3.org/TR/owl-xmlsyntax/, 2012. [Online; accessed Feb-2012].

[13] C. Kaner. Improving the maintainability of automated test suites. In *Proc. of Quality Week 1997*, May 1997.

[14] M. Kim, J. Cobb, M. J. Harrold, T. Kurc, A. Orso, J. Saltz, K. Malhotra, and S. Navathe. Efficient regression testing of ontology-driven systems. http://pleuma.cc.gatech.edu/aristotle/pdffiles/kim_techrep11.pdf/, September 2011.

[15] M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov. Ontology versioning and change detection on the web. In *Proc. of EKAW '02*, pages 197–212, 2002.

[16] M. Klein, A. Kiryakov, D. Ognyanov, D. Fensel, and O. L. Sofia. Finding and characterizing changes in ontologies. In *Proc. of ICCM '02*, pages 79–89, 2002.

[17] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. Firewall regression testing and software maintenance of object-oriented systems. *Journal of Object-Oriented Programming*, 1994.

[18] H. K. N. Leung and L. J. White. Insights into regression testing. In *Proc. of ICSM '89*, pages 60–69, Oct. 1989.

[19] C. McCarty, R. Chisholm, C. Chute, I. Kullo, G. Jarvik, E. Larson, R. Li, D. Masys, M. Ritchie, D. Roden, J. Struewing, W. Wolf, and the eMERGE Team. The emerge network: A consortium of biorepositories linked to electronic medical records data for conducting genomic studies. *BMC Medical Genomics*, 4(1):13, 2011.

[20] S. N. Murphy, M. Mendis, K. Hackett, R. Kuttan, W. Pan, L. C. Phillips, V. Gainer, D. Berkowicz, J. P. Glaser, I. Kohane, and et al. Architecture of the open-source clinical research chart from Informatics for Integrating Biology and the Bedside. *AMIA Symposium*, 2007:548–552, 2007.

[21] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. In *Proc. of ICST '11*, pages 21–30, Washington, DC, USA, 2011. IEEE Computer Society.

[22] D. L. M. Natalya Fridman Noy. Ontology development 101: A guide to creating your first ontology. Technical Report KSL-01-05, Knowledge Systems, AI Laboratory, Stanford University, 2001.

[23] S. T. S. D. Network. RESTful Web Services. http://java.sun.com/developer/technicalArticles/WebServices/restful/, August 2006.

[24] N. F. Noy, H. Kunnatur, M. Klein, and M. A. Musen. Tracking Changes During Ontology Evolution. In *In Proceeding of the 3rd International Semantic Web Conference*, pages 259–273, 2004.

[25] N. F. Noy and M. A. Musen. Promptdiff: a fixed-point algorithm for comparing ontology versions. In *Eighteenth national conference on Artificial intelligence*, pages 744–750, 2002.

[26] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proc. of FSE '04*, pages 241–252, Nov. 2004.

[27] J. Pathak, J. Wang, S. Kashyap, M. A. Basford, R. Li, D. R. Masys, and C. G. Chute. Mapping clinical phenotype data elements to standardized metadata repositories and controlled terminologies: the emerge network experience. *JAMIA*, 18(4):376–386, 2011.

[28] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, Apr. 1997.

[29] B. Smith, M. Ashburner, C. Rosse, J. Bard, W. Bug, W. Ceusters, L. J. Goldberg, K. Eilbeck, A. Ireland, C. J. Mungall, N. Leontis, P. Rocca-Serra, A. Ruttenberg, S.-A. Sansone, R. H. Scheuermann, N. Shah, P. L. Whetzel, and S. Lewis. The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration. *Nat Biotech*, 25(11):1251–1255, 2007.

[30] M. Tury and M. Bieliková. An approach to detection ontology changes. In *Proc. of ICWE '06*, 2006.

[31] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on text differencing. In *Proc. of ENCRESS '97*, pages 3–21, May 1997.

[32] L. J. White and H. K. N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proc. of ICSM '92*, pages 262–270, Nov. 1992.

[33] D. Willmor and S. M. Embury. A safe regression test selection technique for database-driven applications. In *Proc. of ICSM '05*, pages 421–430, 2005.