STRUCTURE COMPARISON AND SEMANTIC INTERPRETATION OF DIFFERENCES*

Wellington Yu Chiu
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, California 90291

ABSTRACT

Frequently situations are encountered where the ability to differentiate between objects is necessary. The typical situation is one in which one is in a current state and wishes to achieve a goal state. Abstractly, the problem we shall address is that of comparing two data structures and determining all differences between the two structures. Current comparison techniques work well in determining all syntactic differences but fall short of addressing the semantic issues. We address this gap by applying AI techniques that yield semantic interpretations of differences.

I INTRODUCTION

One frequently encounters situations where the ability to differentiate between objects is necessary. The typical situation is one in which one is in a current state and wishes to achieve a goal state. Such situations are encountered under several guises in our transformation based programming system research [1, 2, 3]. A simple case is one in which we are given two program states and need to discover the changes from one to the other. Another case is one in which a transformation we wish to apply to effect a desired change does not match in the current state and one wishes to identify the differences. An extension of this second case is the situation where a sequence of transformations, called a development, is to be applied to (replayed on) a slightly different problem.

Abstractly, the problem we shall address is that of comparing two data structures and determining all differences between the two structures. Current comparison techniques work well in determining all syntactic differences but fall short of addressing the semantic issues. In the replay and state differencing situations, comparisons must be more semantically oriented. We address this gap by applying AI techniques that yield semantic interpretations of differences.

This paper describes a part of my thesis: the design of a semantic based differencer and its ongoing implementation.

* This work was supported by National Science Foundation Grant MCS 7683890. The views expressed are those of the author.

II AN EXAMPLE

The following example is presented to show the types of information used to infer and deduce the semantic differences. Below are the before and after program states from a transformational development [1].

```
BEFORE:
while there exists character in text do                    1
   if character is linefeed                                2
      then replace character in text by space;             3
while there exists character in text do                    4
   if P(character)                                         5
      then remove character from text;                     6

AFTER:
while there exists character in text do                    a
begin                                                      b
   if character is linefeed                                c
      then replace character in text by space;             d
   if character in text then                               e
      if P(character)                                      f
         then remove character from text                   g
end;                                                       h
```

The after state above was produced from the before state via application of a transformation but the following explanations of differences were generated without knowledge of the transformation.

- The current syntactic differencing techniques [4, 5, 6, 7, 8, 9, 10] typically explain differences in the following terms:

```
For BEFORE:
        Delete while in line 4
        Delete there in line 4
        . . .
For AFTER:
        Insert if in line f
        Insert then in line f
        . . .
```

- A higher level syntactic explanation is achieved by generalizing and combining the techniques for syntactic differencing to explain differences in terms of embeds, extracts, swaps and associations, in addition to inserts and deletes, and by incorporating syntactic information about the structures being compared [3].

```
The second loop is coerced into
    a conditional.
```

The conditional is embedded into the first loop.

- The proposed explanation of the semantic difference is:

Loops merged.

The following is the derivation of the syntactic explanation. It is presented to show the mechanisms upon which the semantic differencer will be based.

- Syntactically, 2,3 (first loop body of the before state) is equivalent to c,d (part of the loop body of the after state) and 5,6 is equivalent to f,g.

- Infer composite structure 1,2,3 similar to composite structure a,b,c,d,e,f,g,h based on 2,3 being equivalent to c,d.

- Infer an embed of composite structure 4,5,6 into the composite structure 1,2,3 to produce a,b,c,d,e,f,g. The support for this inference comes from 5,6 being equivalent to f,g, the adjacency of 1,2,3 to 4,5,6, and the adjacency of c,d to f,g.

- Infer coercion of loop 4,5,6 to conditional e,f,g based on 5,6 being equivalent to f,g, and the similarity of the loop generator to the conditional predicate.

- Conclude second loop embedded in the first loop.

Our current syntactic differencer produces this type of difference explanation. It extends the techniques currently available by imposing structure on the text strings being compared, thereby making use of structural constraints and the extra structural dimensions in its analysis [3]. Despite this advance, the explanations fall short of the desired semantics of the changes. In the SCENARIO section below, an explanation from the proposed semantic based differencer is presented.

The domain we are dealing with is one where changes are made for the following reasons:

1. To optimize code

2. To prepare code for optimization.

Within such a domain, we can use the constraints on the semantics of changes to derive the semantic explanation "loop merged" and at the same time rule out the syntactic explanation. Building on the mechanisms that generated the derivation above, the following is a proposed derivation of the comparison that yield the semantic interpretation "loops merged".

- Syntactically, 2,3 (the first loop body) is equivalent to c,d and 5,6 (the second loop body) is equivalent to f,g. The context trees (i.e. super trees) containing 2,3 and 5,6 and the context tree for c,d,f,g are the same. Infer FACTORING of context trees with supports being the 2 to 1 mapping of substructures and the equivalence of context trees.

- Infer loop merge from the fact that the context trees for 2,3, 5,6 and c,d,f,g are loop generators.

- Infer the similarity of 5,6 to e,f,g from the syntactic equivalence of 5,6 to f,g. 5,6 embedded in a test for generator consistency inferred from semantic knowledge of loop merging.

- Conclusion: Loops merged.

The explanation generated by our syntactic differencer is not plausible because it doesn't make sense to transform a loop into a conditional only to embed this new conditional into a similar loop. The following is the desired explanation: The body of the second loop is embedded in a conditional that tests for loop generator consistency. This is done without changing the functionality of the body. The two adjacent loops can now be merged subject to any side effects, caused by the first loop body, that will not be caught by the loop generator consistency check around the second loop body.

III DESIGN OF THE SEMANTIC BASED DIFFERENCER

We start by defining relations (profiles) on objects, where objects are the substructures of the structures we are comparing (see Appendix A). The information provided by this profile consists of:

- Sequence of nonterminals from the left hand side of productions used in generating the context tree of the substructure. A context tree (i.e. super tree) is that part of the parse tree with the substructure part deleted.

- The sequence of productions used in the derivation of the substructure, given the context tree above as the starting point.

- Positional information.

Our syntactic differencer makes use of the information provided by the object profile to determine base matches. The techniques used in the differencer to determine base matches are combinations of the common unique, common residual and common super tree techniques described in [3, 4, 6, 7, 10]. Below is a brief description of the common unique and common residual techniques for linear structures.

- Longest Common Subsequence (LCS): The main concern with LCS is that of finding the minimal edit distance between two linear structures. The only edit operations allowed are: delete a substructure, or insert a substructure [6, 8]. For the above example, the LCS is: 1,2,3,5,6 matching a,c,d,f,g.

- Common Unique (CU): The key to this technique the use of substructures that are common to both structures and unique within each as anchors [4]. For the above example, the common unique substructures are: linefeed, replace, space, P, remove.

We then build on these base matches by inferring matches of substructures that contain substructures that are matches. An example of this is inferring that 1,2,3 is similar to a,b,c,d,e,f,g,h

from the assertion that 2,3 matches c,d. There are two types of inferred matches: those without syntactic boundary conflicts and those with conflicts. Syntactic boundary conflicts result from embedding, extracting or associating substructures.

The third type of profile is one that describes the relationship between substructures within a given structure. Considerations here are: adjacency, containment, and relative positioning.

There are several semantic rules that describe a majority of structure changes. Some are: factoring, distributing, commuting, associating, extracting, embedding, folding, and unfolding. A partial description of the factor semantics can be found in Appendix A. Factors currently considered by our semantic rules are: support for matches, the generating grammar, object profiles, and relations between substructures of the same structure. With each set of examples tried, we add to our set of semantic rules, and our intuitive guess, given our domain of changes due to optimization or preparation for optimization, is that this set will be fairly small when compared to the set of transformations needed in a transformation based programming system.

IV A SCENARIO

Our syntactic differencer makes use of structural information. For LISP programs it know about S-expressions. For programs written in our specification language, differencing is performed on the parse trees. The differencer first tries to isolate differences into the smallest composite substructure containing all changes. With this reduced scope, the differencer uses the common unique technique to assert relations on content base matches. In our example, substructure 2,3 is equivalent to c,d and substructure 5,6 is equivalent to f,g.

Once all of these possible assertions have been made, we use them as anchors to assert relations based on positional constraint and content matches (residual structure matches). This residual technique works well as a relaxation of the uniqueness condition in the common unique requirement, and acts as a backup in case no substructures are common to both structures and unique within each. The super tree technique is used as a default when neither of the above techniques applies. The intuitive explanation for this third technique has to do with both objects occupying the same position with respect to a common super tree. With the super tree technique, content equivalence is relaxed.

With the two asserted relations regarding substructures 2,3 being equivalent to c,d and 5,6 being equivalent to f,g, we now infer that 4,5,6 is similar to e,f,g because 5,6 is common unique with f,g and without conflicting evidence (i.e. boundary violations) the assertion is made. Once made, further analysis of this reduced scope shows relationships between the loop generator 4 and the conditional predicate e.

Since we are given, via the super tree technique, that 1,2,3,4,5,6 matches a,b,c,d,e,f,g,h we assert the inference 2,3 to c,d even though conflicts due to boundary violations arise. The boundary violation in this instance is the mapping of two substructures into one (i.e. the segmental or n-m problem). Given that we want to produce all plausible explanations, we assert that 1,2,3 is similar to a,b,c,d,e,f,g,h because 2,3 and c,d are common unique matches. With this assertion, we could with our Embed Semantics say that the second loop is embedded in the first. But our knowledge about optimizations makes more plausible the Factor Semantics that is triggered by the segmental matching.

When the Factor Semantics is triggered, the relationships within a given structure, such as adjacency and relative positioning, are asserted (see Appendix A). All the requirements except for body2 being equivalent to body4 are met. But once the cases are considered, we discover that the operator being factored is indeed a loop generator and that we can relax the requirement that body2 be equivalent to body4 to that of similarity of the two bodies. This follows from the support for the relationship between body2 and body4. Final analysis reveals that body2 is embedded in body4.

V CONCLUSION

Given the small example above, we see that the derivation of the semantic difference involves syntactic and semantic knowledge of the structure domain as well as techniques for managing and applying the knowledge. We present a design that addresses the issue of managing and applying both syntactic and semantic knowledge of the structures being compared so as to provide a semantic interpretation of changes. This allows us to bridge the gap that exists between the information provided by current differencers and the information needed by current differencing tasks.

VI APPENDIX A: TEMPLATES FROM THE SEMANTIC DIFFERENCER

Every substructure of the structures being compared has associated with it an Object Profile that is an record with the following role names as fields:

Content:	Value of the substructure.
Type Context:	Sequence of nonterminals of productions, of the grammar, used in the derivation of the substructure.
Positional Context:	Position in parse tree. (i.e. a sequence of directions in reaching the substructure from the root of the parse tree.
Abstraction:	If a grammar is used to generate the substructure, this refers to the sequence of productions used to generate the substructure itself.

A Relation Profile describes the relationships between two substructures, one from each of the structures being compared. The role names of this record are:

```
Base        Content (i.e. common unique)
Matches:    Context (i.e. positional determined
                from context trees)

            Positional constraint and Content
                (i.e. from the largest common
                residual substructure technique
                where uniqueness of context
                matches is relaxed).

Inferred    Conflict free (i.e. no syntactic
Matches:        boundary violations)

            With conflicts (inferences depends on
                heuristics regarding current
                substructure abstraction and
                weights associated with
                substructure matches).

Mappings: 1-1 substructure matches
          2-1 substructure matches
          n-m substructure matches
```

A second relation profile is one between substructures within a given structure. Some of the considerations here are: adjacency of two substructures, containment, and relative positioning.

There are several semantic rules that describe a majority of structure changes. Some are factoring, distributing, commuting, associating, extracting, embedding, folding and unfolding. Below is a partial description of the Factor Semantics used in generating the semantic interpretation above:

```
FACTOR SEMANTICS:
        FORM:   LHS:    op1 body1
                        op2 body2
                RHS:    op3 body3 body4

        KEY:    Segmental matching

        REQUIREMENTS:
        requirements    op1=op2
        requirements    op1=op3
        requirements    body1=body3
        requirements    body2=body4
        requirements    relative positioning of
                        body1 to body2 holds for
                        body3 to body4
        requirements    adjacency of body1 to
                        body2 holds for body3 to
                        body4

        CASES:
        op1 is a loop generator
                relaxations  body2=body4 relaxed to
                             body2 similar to body4

        RELAXATIONS:
        relaxations     adjacency requirement
        relaxations     relative positioning
        relaxations     equivalence (=) to similar
```

VII REFERENCES

1. Balzer, R., Goldman, N., and Wile, D., "On the Transformational Implementation Approach to Programming," in *Second International Conference on Software Engineering*, pp. 337-344, IEEE, October 1976.
2. Balzer, R., *Transformational Implementation: An Example*, Information Sciences Institute, Research Report 79-79, September 1979.
3. Chiu, W., Structure Comparison, 1979. Presented at the Second International Transformational Implementation Workshop in Boston, September, 1979.
4. Heckel, P., "A Technique for Isolating Differences Between Files," *Communications of the ACM* 21, (4), April 1978, 264-268.
5. Hirschberg, D., *The Longest Common Subsequence Problem*, Ph.D. thesis, Princeton University, August 1975.
6. Hirschberg, D., "Algorithms for the Longest Common Subsequence Problem," *Journal of the ACM* 24, (4), October 1977, 664-675.
7. Hunt, J., *An Algorithm for Differential File Comparison*, Bell Laboratories, Computer Science Technical Report 41, 1976.
8. Hunt, J., "A Fast Algorithm for Computing Longest Common Subsequences," *Communications of the ACM* 20, (5), May 1977, 350-353.
9. Tai, K., *Syntactic Error Correction in Programming Languages*, Ph.D. thesis, Cornell University, January 1977.
10. Tai, K., "The Tree-to-Tree Correction Problem," *Journal of the ACM* 26, (3), July 1979, 422-433.