Introduction

Incremental Techniques store the results of previous computations and use them constructively for succeeding computations. The inclusion of incrementality in any system today is prompted more out of necessity than as an additional feature. Incremental systems allow facilities for editing, drawing, modelling, etc., and provide enough scope for the user to experiment with the system. In order to be interactive and more user-friendly, the system should be able to provide quick results, almost instantaneously, to any user-initiated modification. Quick response times may be achieved by making algorithms even faster. This, however, is limited to their optimal performance. The other clear option is to transform the algorithms into their incremental versions.

The Structure Oriented Software Development Environments (SOE) were one of the earliest systems where the need for incrementality was felt. SOEs are a class of programming environments which, by integrating various Software Development activities, strive to overcome the lacunae in the traditional Environments. This is achieved by making the development process very interactive. The initial motivation for SOEs was to provide the user with an interactive tool -- a syntax directed editor -- for entering software product, i.e., requirement specification, design specification, program, etc., in terms of language constructs. This capability was extended by combining the editor with the classical analysis phase of the compiler. An SOE, therefore, reinforces interactively, it's knowledge of syntax and semantics of language during the development process.

*Attribute Grammars* have proved to be a good theoretical basis for developing SOEs. They allow the specifications of the syntactic and the semantic properties of a software product to be defined very clearly and precisely. Each product is represented as an
attributed derivation tree. However, the number of attribute instances generated during the development process may be extremely large. As the reevaluation of the attributes needs to be incremental in nature, to provide an immediate feedback to the developer, the attribute evaluation algorithm becomes the most crucial component of an SOE. The acceptability of an SOE is, therefore, determined by the efficiency of its incremental evaluation algorithm.

Techniques for incremental evaluation of attributes are broadly categorized into those based on AG formalism and those based on any other ad-hoc or generalised scheme. The AG based evaluators are further classified into static and dynamic evaluator schemes. The static evaluators can be built only after a thorough analysis of the particular AG. The static scheme presumes information that arise out of the dependencies between various attributes belonging to a production rule of an AG. The static evaluators are generally faster, but a thorough analysis of the AG is required at the evaluator construction time. It is almost impossible, to construct evaluators for a generalised AG, as it is an NP complete problem. However, a host of static evaluators have been proposed for subclasses of AGs, where it is sometimes guaranteed that the evaluators can be constructed in polynomial times.

Dynamic Evaluators, on the other hand, have no such restrictions and most of them can work for any class of AGs. However, the majority operates under the constraint that there should be no cycles in the dependency graph constructed from the attributes and their defining equations.

Attribute Grammar based evaluators have been used as a basis in a variety of other applications including language compilers, compiler generators, program transformation systems, interface management systems, natural language processors, etc.
Introduction

*Incremental Dynamic Evaluators*, restore an inconsistent dependency graph to its consistent state after any graph modification, representing some change in the underlying substrate. The set *Directly Modified* is the set which has actually undergone changes. *INFLUENCED* is the set of vertices which are directly or transitively dependent on *Directly Modified*. This denotes a set of vertices which might be potentially inconsistent. However, it is possible that all these vertices may not change. *AFFECTED* is the set of vertices which actually change values.

Any evaluator will be classified as optimal if it achieves two targets:
1) the optimal reevaluation set and,
2) the optimal overhead complexity.

The optimal reevaluation set is *AFFECTED*, the set of all the vertices whose values are different from those in the final consistent graph. Any evaluator has an optimal overhead complexity if it takes no more than $O(|AFFECTED|)$ time for book keeping. The few solutions that have been suggested for solving the problem have some drawbacks: some of them evaluate attributes outside the set *AFFECTED*, or the space complexity is high, as they require maintenance of large data structures called characteristic graphs. Further, they impose constraints like single edit cursor movement and are applicable to only subclasses of AGs.

In this thesis, we propose an algorithm which is applicable to all classes of non-cyclic AGs. In addition, no restriction on the type of modification is imposed. Modifications may be effected in widely separated areas of the dependency graph before invoking the evaluation algorithm. The algorithm evaluator computes the set *AFFECTED* in time $O(|INFLUENCED|)$. Even though the book keeping time here is suboptimal, no extra data structures need to be maintained.
Another peculiarity to be noticed in Attribute Grammar based dependency graph is the occurrence of large copy rule chains. These are implicit identity functions, used to communicate attribute values from one portion of the attributed semantic tree to another. Copy rule chains normally occur in the form of trees as a vertex may be the head for a number of such chains. Therefore, values need to be communicated from the root of a copy rule tree to all its leaves, bypassing all the intermediate vertices.

Hoover[Hoo87], introduced the concept of structure trees. These are structure preserving projections of copy trees and the number of vertices in them is of the order of the vertices to which the attribute values need to be finally communicated. But Hoover's algorithm takes $O(n)$ overhead to maintain the structure trees in the face of any modification, where $n$ is the total number of copy vertices in the system.

If copy trees are maintained as trees of paths, then Sleator and Tarjan's[ST83,ST85] technique of self adjusting binary search trees can be applied to give an amortised bound of $O(\log n)$ for maintaining the copy trees and their associated structure trees. We propose algorithms `add_edge` and `delete_edge` for effecting edge addition or deletion in a dependency graph which maintains copy trees as self adjusting trees.

The dependency graph, which dynamically represents the attribute dependencies, can be built from the substrate in an incremental manner by adding or deleting vertices, edges, or components after invoking the appropriate procedures. If vertices, having copy rules as their semantic equations are detected, they are included in their adjoining copy trees or new trees are created. Further, all the copy trees in the graph are maintained as self-adjusting search trees.

$AFFECTED_c$ is the set of all vertices in $AFFECTED$ after removing the redundant copy vertices. Similarly, $INFLUENCED_c$ is a subset of $INFLUENCED$ obtained after removing
all the copy vertices from \textit{INFLUENCED}. The proposed algorithm \textit{Restore\_Consistency}, is a \textit{dynamic incremental evaluator}, with the time complexity of $O(\text{EVAL}(\text{AFFECTED}_c) + |\text{INFLUENCED}_c|)$. \textit{Restore\_Consistency} follows a topological order for evaluation, bypassing all the copy vertices at the same time. The only input to the algorithm \textit{Restore\_Consistency} is \textit{Directly Modified}, the set of all vertices which have been modified as a consequence of some operation in the substrate. The algorithm is quite generalised and can even be equally used by the systems other than those based on AGs. Since the algorithm does not maintain any additional data structure, except the structure trees, (which are part of the copy trees), its space complexity is better than any existing algorithm. Its time complexity is equal to other existing algorithms, in terms of the reevaluation set, but the book keeping time is suboptimal. Further, the proposed algorithm can be used for unconstrained dependency graph modifications. No existing algorithm which allows unconstrained modifications has ever achieved the optimal book keeping time.

**Outline of the Thesis**

Structure Oriented Environment which allows programs to be developed in terms of their syntactic constructs, were the first ones to pose the problem of incremental attribute evaluation. In Chapter 2, we discuss in detail these environments and the need for incrementality in all aspects of their operation.

Chapter 3 can be read separately from the rest of the thesis. It introduces the idea of incrementality and gives a survey of some of the prevalent techniques used for achieving incrementality into systems.

Attribute Grammars, which form the theoretical foundation for most \textit{SOEs} and other systems like compilers, Optimisers etc. are introduced in Chapter 4. Incremental Attribute Evaluators techniques are also introduced. The problem is approached through a process...
of stepwise refinement. A naive evaluator algorithm is introduced and is continuously refined to give algorithm Evaluate2 which has a time complexity of $O(|INFLUENCED|) + EVAL(AFFECTED))$.

Chapter 5 introduces the concept of copy trees and the manner in which copy trees and their associated structure trees may be embedded as self-adjusting binary search trees. Maintenance of a dependency graph which include copy trees, in the face of incremental modification is discussed in Chapter 6 and algorithms proposed for different situations. Further, algorithm Restore_Consistency, is presented, for restoring a dependency graph to its consistent state. Finally, Chapter 7 presents a summary of the ideas discussed in this thesis, draws some conclusions, and suggests some areas for future research.