Supporting the Software Development Process with
Attributed NLC Graph Grammars

Simon M. Kaplan
Steven K. Goering
Roy H. Campbell

Department of Computer Science
University of Illinois
Urbana, Illinois 61801, USA

ABSTRACT. An important research problem in software engineering is to find appropriate formalisms
and tools to support the software development process. Efforts to build program development support tools
have developed schemes that employ an internal data structure based on trees. Trees are inherently limited
and also create long path lengths along which semantic information is forced to flow. We propose ANLC
graph grammars as a formalism which can be used to generate tools based on graphs rather than tree
structures. This paper defines ANLC grammars, illustrates the use of the formalism with an example and
discusses the advantages of the use of graphs rather than trees in building program development tools.

Key Words: Attribute, Graph, Graph Editor, Graph Grammar, Neighbourhood Controlled Embedding,
Node Label Controlled, Programming Environment, Software Support Environment.

CONTENTS

1. Introduction
2. NLC Graph Grammars
3. Attributed NLC Grammars
4. Using ANLC Graph Grammars
5. Example
6. Supporting the Software Development Process
7. Related Work
1 Introduction

The development of tools that will assist in all aspects of the software development process is a major software engineering research problem. This paper introduces attributed NLC (ANLC) graph grammars, a formalism that we believe is suitable for the specification of tools that support many different aspects of the software development process.

Traditional program development tools that are generated from formal specifications [15] [4] [16] all use trees as the internal structure to represent the object (usually a program) that a user is editing. The tree corresponds to the derivation tree of the string representing the program. An important feature of such tools is that they incrementally (i.e. after each edit) check the semantic consistency of the structure, thereby providing the user with a programming environment in which errors in his program are reported immediately. This incremental checking is performed by passing attribute values that represent semantic information along arcs of the tree. After an edit on the tree, the appropriate set of attributes is reevaluated so that the semantic consistency of the program can be reestablished. From the viewpoint of the software researcher building such environments, the derivation tree representing a program is far more important than the string itself, and most research into programming environments has concentrated on manipulating this tree, with the string representing the program as a byproduct of this manipulation.

In order to generate such tools automatically, attribute grammars [13] are most commonly used. An attribute grammar is a context-free grammar which describes the structure of legal derivation trees augmented with attribution rules which describe the semantic restrictions on the grammar. Algorithms to incrementally perform attribute updating after an edit in optimal time have been developed by Reps [16].

Many interesting structures such as module interdependence structures, manager/programmer relations or critical path networks for which one might want to develop editing systems are graphs rather than trees. We believe that in order to automate the software development process, it must be possible to build environments that manipulate graph structures rather than trees. We must also retain the utility of editing on trees, specifically the ability to perform incremental semantic consistency checking. Further, we do not want to build tools for individual applications by hand, as this would be far too expensive; rather we wish to build a tool generating system that can take as input a specification of a tool and generate a corresponding tool automatically. We plan to use graph grammars as the formalism in which to specify the graph structures. Because we want to permit semantic analysis of programs represented by these structures, we will augment the graph grammars with attribution rules to obtain attributed graph grammars. In order to manage the complexities inherent in graph rewriting systems, we use node-label controlled (NLC) graph grammars [10].

This paper overviews attributed NLC graph grammars and illustrates their use through a simple example. The body of the paper is structured as follows. Section 2 introduces NLC grammars. Section 3 introduces ANLC grammars and overviews some results concerning properties of the graphs. Proofs of results are omitted due to lack of space. Section 4 discusses editing on graphs, and incrementally updating semantic information after an edit on a graph. Section 5 illustrates the use of the formalism with an example and section 6 elaborates on some practical applications of the formalism. The paper concludes with a discussion of related work.