Incremental Attribute Evaluation for Multi-User Semantics-Based Editors

Josephine Micallef

Columbia University
Department of Computer Science
New York, NY 10027

May 1991

CUCS-023-91

Incremental Attribute Evaluation for Multi-User Semantics-Based Editors

Josephine Micallef

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences.

Columbia University
1991

Abstract

Incremental Attribute Evaluation for Multi-User Semantics-Based Editors

Josephine Micallef

This thesis addresses two fundamental problems associated with performing incremental attribute evaluation in multi-user editors based on the attribute grammar formalism: (1) multiple asynchronous modifications of the attributed derivation tree, and (2) segmentation of the tree into separate modular units. Solutions to these problems make it possible to construct semantics-based editors for use by teams of programmers developing or maintaining large software systems. Multi-user semantics-based editors improve software productivity by reducing communication costs and snafus.

The objectives of an incremental attribute evaluation algorithm for multiple asynchronous changes are that (a) all attributes of the derivation tree have correct values when evaluation terminates, and (b) the cost of evaluating attributes necessary to reestablish a correctly attributed derivation tree is minimized. We present a family of algorithms that differ in how they balance the tradeoff between algorithm efficiency and expressiveness of the attribute grammar. This is important because multi-user editors seem a practical basis for many areas of computer-supported cooperative work, not just programming. Different application areas may have distinct definitions of efficiency, and may impose different requirements on the expressiveness of the attribute grammar. The characteristics of the application domain can then be used to select the most efficient strategy for each particular editor.

To address the second problem, we define an extension of classical attribute grammars that allows the specification of interface consistency checking for programs composed of many modules. Classical attribute grammars can specify the static semantics of monolithic programs or modules, but not inter-module semantics; the latter was done in the past using *ad hoc* techniques. Extended attribute grammars support programming-in-the-large constructs found in real programming languages, including textual inclusion, multiple kinds of modular units and nested modular units. We discuss attribute evaluation in the context of programming-in-the-large, particularly the separation of concerns between the local evaluator for each modular unit and the global evaluator that propagates attribute flows across module boundaries. The result is a uniform approach to formal specification of both intra-module and inter-module static semantic properties, with the ability to use attribute evaluation algorithms to carry out a complete static semantic analysis of a multi-module program.

Table of Contents

List of Figures

Acknowledgements

I would like to thank my advisor, Gail Kaiser. She introduced me to this research area, provided thoughtful guidance throughout the development of this work, and pointed out the broader context for applying my research. I wish to thank Dan Yellin for his insightful readings of my work. Yechiam Yemini deserves special credit for his support during the years. I am also grateful to Terry Boult and Barbara Ryder for serving on my committee, and to Stu Feldman for encouraging my research during my summer at Bellcore.

I wish to thank the many people who have contributed to the MERCURY project, in particular, Travis Winfrey, Yael Cycowicz, and Wen-wey Hseush. I also thank Naser Barghouti, Mark Gisi and other Frodo group members for useful comments about my research, and to Chris Maio for providing assistance with our computer systems.

My years at Columbia would not have been as enjoyable if it were not for the many friends I found there. Besides being a wonderful friend, Dannie Durand lent her support over the years. My two office mates and friends, Monnett Hanvey and Dorèe Seligmann, made our office a much more pleasant environment to work in.

I am grateful for the financial support provided by the American Association of University Women during my last year at Columbia.

Finally, I would like to acknowledge the love and support I received from my husband George and my son David throughout the sometimes trying years I have been in graduate school. I also am grateful for my baby Michelle who provided welcome interruptions from writing during the last two months. I am indebted to my mother for coming from Malta to help out while I completed the thesis.

To George, David, and Michelle

Chapter 1
Introduction

1.1. Motivation

The problem of incremental attribute evaluation has been the focus of much research in the last few years. Incremental attribute evaluation algorithms are of practical importance in semantics-based editors — also known as language-based editors — that are based on the attribute grammar formalism [Knuth 68]. With semantics-based editors, users construct or modify their programs by selecting from a menu of legal templates, entering by hand identifiers, strings, integers, *etc.* The editor guarantees syntactic correctness and aids the user in writing semantically correct programs by incrementally checking for static semantic errors after every editing command, and highlighting any inconsistencies so the user can immediately locate and understand the problem while still in context.

Semantics-based editors are well-understood tools for educating student programmers and reminding casual programmers of the programming language rules [Garlan 84, Chandhok 85]. To date, however, semantics-based editors have rarely been used by advanced students or industry programmers, due to two basic inadequacies. First, the template-based user interface is deemed unacceptable by many experienced programmers. Second, the editors available prior to this work support only an individual programmer working on a small program or a single module of a large system, and do not help at all with detecting inconsistencies among modules written by different programmers [Perry 85, Perry 87]. There has been much work on solving the first problem, with conventional text editing user interfaces tied to the structure editor through incremental parsing technology (*e.g.*, [Wegman 80, Morris 81]) and customization of the user interface based on user modeling (*e.g.*, [Neal 87]). This thesis addresses the second problem.

The new results developed in this thesis advance the attribute grammar technology to support *multi-user* semantics-based editors [Kaplan 86]. Multi-user editors would enable collaboration among multiple student programmers working together on a group project, and more significantly, support for programming-in-the-many for commercial software development. Further, multi-user semantics-based editors could promote collaboration on other kinds of structured documents, besides programs, and thus provide a practical basis for many areas of computer-supported cooperative work [Greif 88].

1.2. Thesis Problem

The focus of the thesis is on the algorithms and facilities needed to make multi-user semantics-based editors work. We extend the basic technology commonly employed for single-user semantics-based editors: representation of the program (or other structured document) as an attributed derivation tree, modeling of edits as subtree replacements in this tree, specification of the syntactic and semantic properties of the programming language (or other application domain) with an attribute grammar, and immediate propagation of changes introduced by an edit through incremental evaluation of the attributes affected by the edit (*e.g.*, [Johnson 82, Reps 84a]).

We address two fundamental problems associated with performing incremental evaluation of attribute grammars in multi-user editors:

1. *Multiple asynchronous modifications to the attributed derivation tree.*
2. *Segmentation of the derivation tree into separate modular units.*

In the first problem, we are concerned with how to correctly and efficiently carry out incremental attribute evaluation in response to asynchronous changes made by multiple users. Previous algorithms for updating attributes in response to single or multiple synchronous edits [Reps 83, Yeh 83, Reps 86, Yeh 88] are not effective for multi-user edits, either requiring the users to ``take turns'', or to wait until several changes have been made before receiving feedback about their change.

In the second problem, the question is how to extend the attribute grammar formalism and attribute evaluation techniques to support programming-in-the-large constructs found in real programming languages, such as multiple kinds of compilation units, nested compilation units, and textual inclusion.

1.3. Application Scenarios

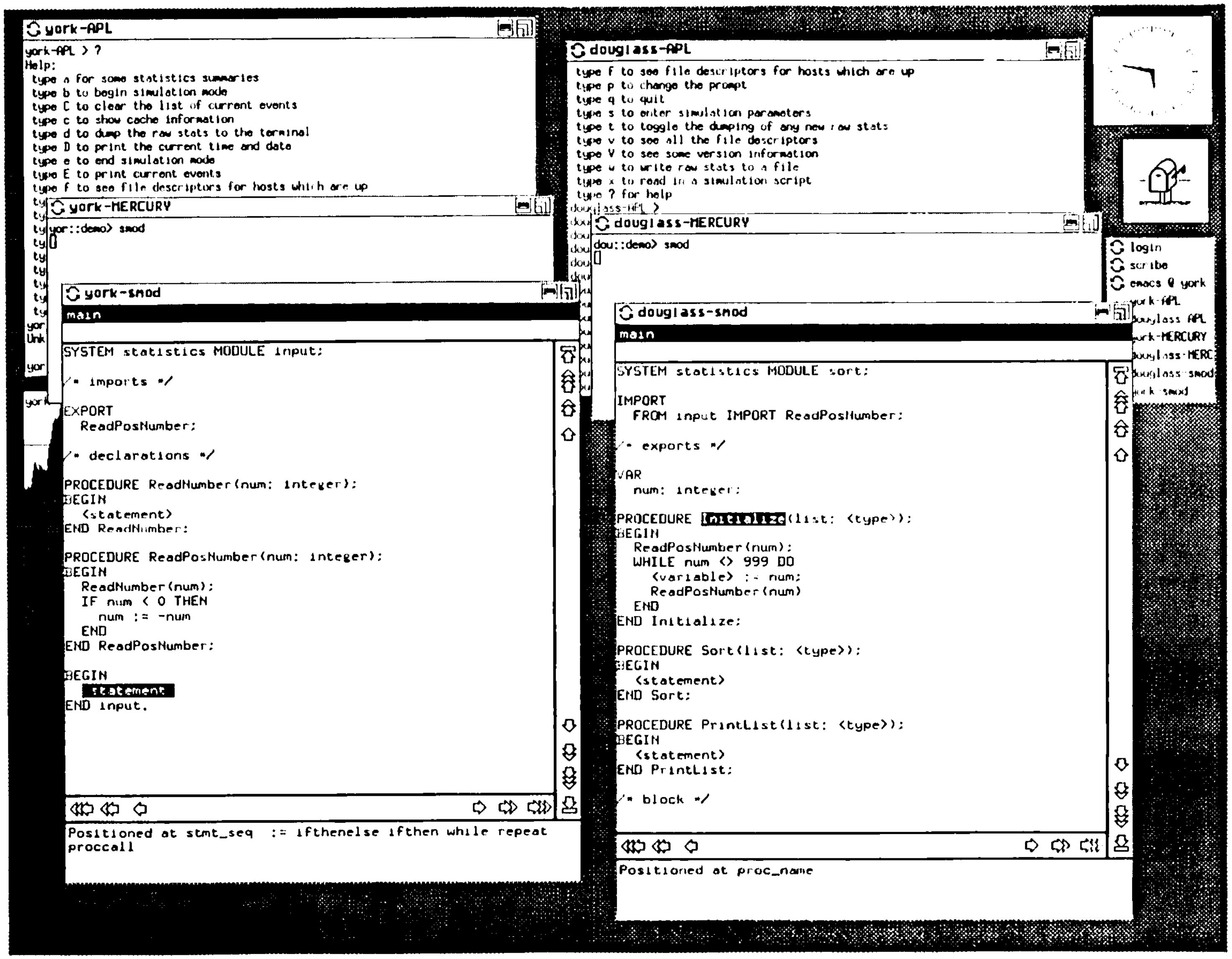
The technology developed in this thesis is applicable to a wide variety of applications that are based on attribute grammars. Potential applications have the following characteristics:

- The representation of the central data objects of the application is an attributed tree.
- The tree can be modified by multiple asynchronous subtree replacements initiated by external agents (humans or automated transformations).
- The tree is partitioned into segments according to the granularity of access rights with respect to these agents.

Below we describe two distinct application scenarios for incremental attribute evaluation algorithms supporting multiple users, one for programming-in-the-many in the Modula-2 programming language [Wirth 82] and the other a distributed calendar system for scheduling meetings among groups of people. Both of these applications have been implemented in the MERCURY system, which generates distributed multi-user semantics-based editors from attribute grammar specifications. (The MERCURY system is described in appendix A.) Examples in the rest of the thesis are drawn from the programming application domain.

1.3.1. Smod

The Smod environment, for Small Modula-2, supports teams of programmers writing programs in a subset of Modula-2. Smod includes only that portion of Modula-2 necessary to demonstrate consistency checking across module interfaces: import and export statements; the elementary data types INTEGER, REAL, BOOLEAN and CHAR; procedure declarations, restricted to only allow value parameters, and procedure calls; assignment, conditional and repetitive (while and repeat) statements; and expressions.



```
york-APL
york-APL > ?
Help:
 type a for some statistics summaries
 type b to begin simulation mode
 type C to clear the list of current events
 type c to show cache information
 type d to dump the raw stats to the terminal
 type D to print the current time and date
 type e to end simulation mode
 type E to print current events
 type f to see file descriptors for hosts which are up
```

```
york-MERCURY
yor::demo> smod
```

```
york-smod
main

SYSTEM statistics MODULE input;

/* imports */

EXPORT
  ReadPosNumber;

/* declarations */

PROCEDURE ReadNumber(num: integer);
BEGIN
  <statement>
END ReadNumber;

PROCEDURE ReadPosNumber(num: integer);
BEGIN
  ReadNumber(num);
  IF num < 0 THEN
    num := -num
  END
END ReadPosNumber;

BEGIN
  statement
END input.

Positioned at stmt_seq  := ifthenelse ifthen while repeat
proccall
```

```
douglass-APL
 type F to see file descriptors for hosts which are up
 type p to change the prompt
 type q to quit
 type s to enter simulation parameters
 type t to toggle the dumping of any new raw stats
 type v to see all the file descriptors
 type V to see some version information
 type w to write raw stats to a file
 type x to read in a simulation script
 type ? for help
douglass-APL >
```

```
douglass-MERCURY
dou::demo> smod
```

```
douglass-smod
main

SYSTEM statistics MODULE sort;

IMPORT
  FROM input IMPORT ReadPosNumber;

/* exports */

VAR
  num: integer;

PROCEDURE Initialize(list: <type>);
BEGIN
  ReadPosNumber(num);
  WHILE num <> 999 DO
    <variable> := num;
    ReadPosNumber(num)
  END
END Initialize;

PROCEDURE Sort(list: <type>);
BEGIN
  <statement>
END Sort;

PROCEDURE PrintList(list: <type>);
BEGIN
  <statement>
END PrintList;

/* block */

Positioned at proc_name
```

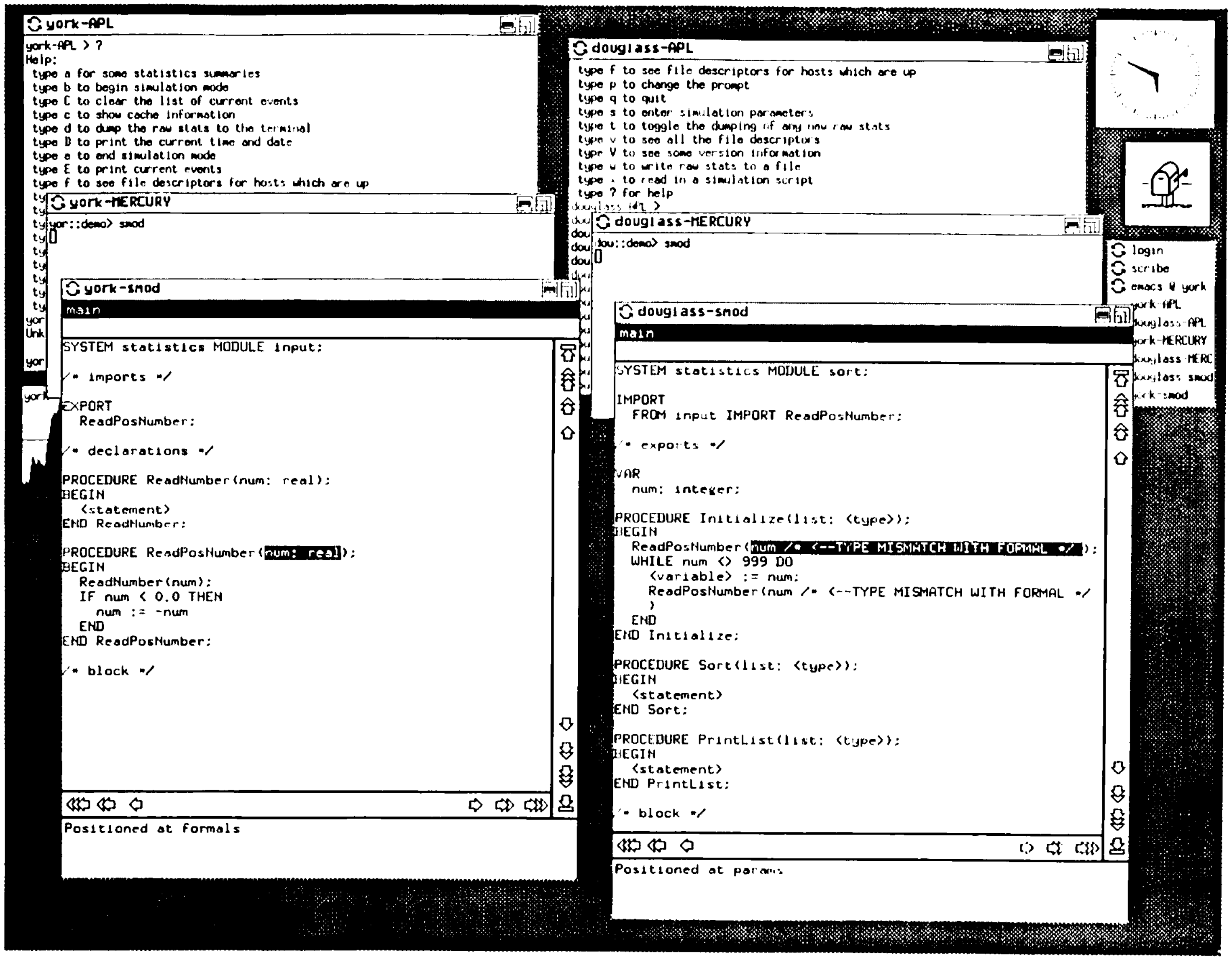
Figure 1-1: *Two Smod modules before Interface Change*

Figure 1-1 shows two windows, each representing a different programmer working on a separate module. The two windows are shown on the same screen for illustrative purposes, but each is connected to a different user process on a different machine (york, a Sun 3/60, and douglass, a Sun 3/280). At this point, the second programmer's module `sort` is correctly using the interface exported by the first programmer's module `input`.

Figure 1-2 shows the same two windows, but after the first programmer has modified the interface to change the `ReadPosNumber` facility exported by `input` and imported by the second programmer into `sort`. The change results in the appearance of two error indications in the second programmer's window, each informing him or her of an inconsistency (where `ReadPosNumber` is used within `Initialize`), immediately after the first programmer completes the keystroke necessary to edit the interface. Thus each programmer is automatically and immediately kept up to date about all program changes made by others that affect him or her.

This is important for programming-in-the-many, where a large program is divided into modules, each assigned to a different programmer. In theory, the module interfaces are set in advance, are ''correct'' and should not change, but in practice module interface errors and their repairs account for a large portion of changes [Perry 85, Perry 87]. In most software development efforts, programmers rely on electronic mail, meetings, and other human-directed mechanisms for informing each other about interface changes, but these are often tardy, incomplete, or misleading — so there may be long periods where programmers make out-of-date or incorrect assumptions about interfaces, resulting in much wasted effort based on these assumptions.

Based on the results of this thesis, MERCURY makes it possible to automate propagation of all interface changes to <u>all</u> affected programmers, and to <u>only</u> those programmers actually affected (to avoid overloading programmers with irrelevant information about interface changes that do not affect them). The goal is to increase programmer productivity and software quality by making sure that programmers always have all the (static semantic) information they need about the other modules that their own work depends on.



```
york-APL > ?
Help:
 type a for some statistics summaries
 type b to begin simulation mode
 type C to clear the list of current events
 type c to show cache information
 type d to dump the raw stats to the terminal
 type D to print the current time and date
 type e to end simulation mode
 type E to print current events
 type f to see file descriptors for hosts which are up
```

```
yor::demo> smod
```

```
SYSTEM statistics MODULE input:

/* imports */

EXPORT
  ReadPosNumber;

/* declarations */

PROCEDURE ReadNumber(num: real);
BEGIN
  <statement>
END ReadNumber;

PROCEDURE ReadPosNumber(num: real);
BEGIN
  ReadNumber(num);
  IF num < 0.0 THEN
    num := -num
  END
END ReadPosNumber;

/* block */
```

```
Positioned at formals
```

```
 type f to see file descriptors for hosts which are up
 type p to change the prompt
 type q to quit
 type s to enter simulation parameters
 type t to toggle the dumping of any new raw stats
 type v to see all the file descriptors
 type V to see some version information
 type w to write raw stats to a file
 type x to read in a simulation script
 type ? for help
douglass-APL >
```

```
dou::demo> smod
```

```
SYSTEM statistics MODULE sort:

IMPORT
  FROM input IMPORT ReadPosNumber;

/* exports */

VAR
  num: integer;

PROCEDURE Initialize(list: <type>);
BEGIN
  ReadPosNumber(num /* <--TYPE MISMATCH WITH FORMAL */);
  WHILE num <> 999 DO
    <variable> := num;
    ReadPosNumber(num /* <--TYPE MISMATCH WITH FORMAL */
    )
  END
END Initialize;

PROCEDURE Sort(list: <type>);
BEGIN
  <statement>
END Sort;

PROCEDURE PrintList(list: <type>);
BEGIN
  <statement>
END PrintList;

/* block */
```

```
Positioned at params
```

Figure 1-2: *Two Smod modules after Interface Change*

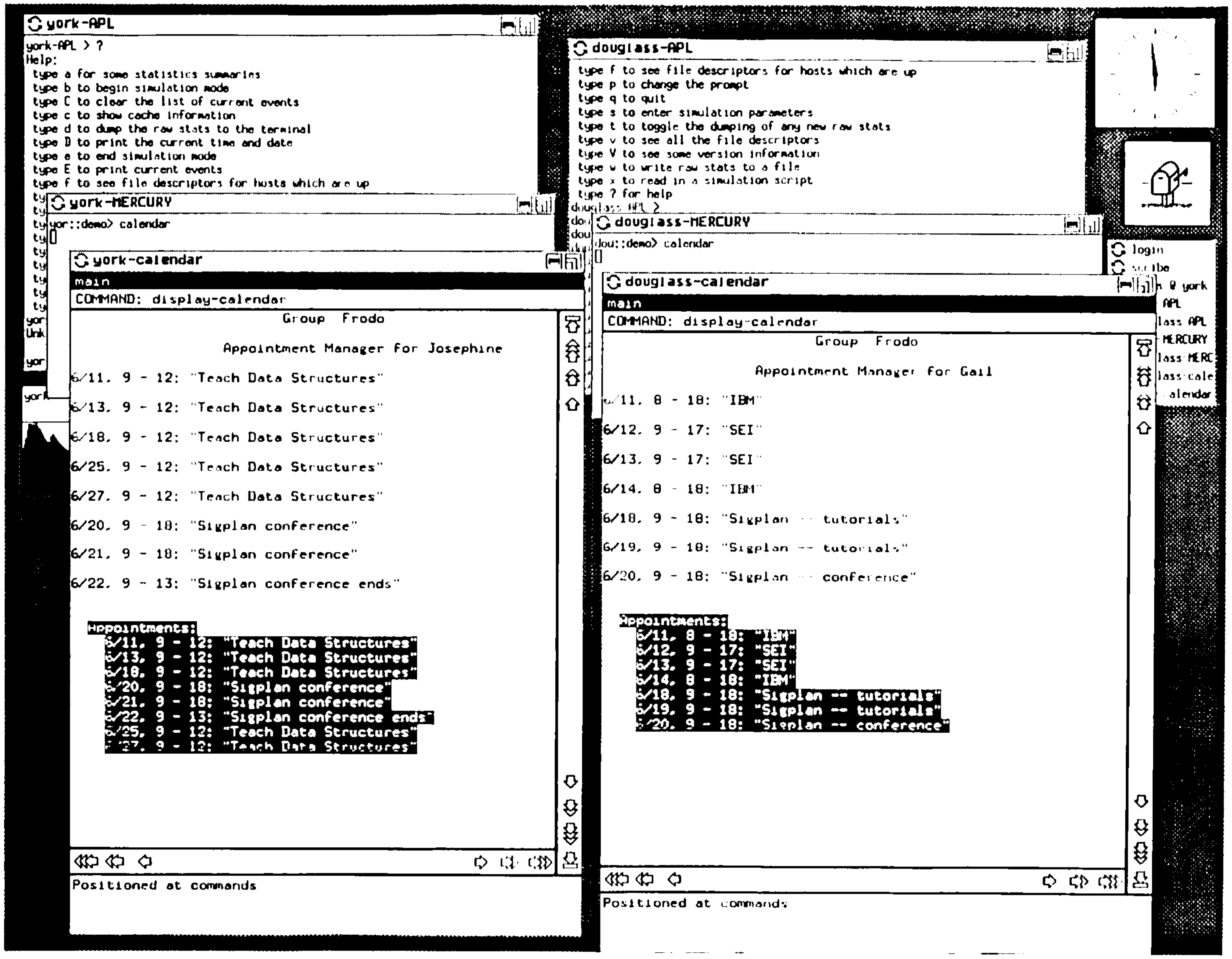
1.3.2. Calendar

The Calendar system is our first significant attempt at a non-programming computer-supported cooperative work application. Although not a full calendar system, it provides most of the automatic scheduling functionality recommended in a study of electronic calendars in office systems [Kincaid 85].

Each user of our system maintains his or her own personal calendar on-line, and enters personal appointments. When one user desires to set up a meeting involving several people, he or she enters a request indicating the constraints for the meeting: who should attend, the expected length of the meeting, the earliest and latest dates for the meeting, and some indication of its purpose. Calendar checks the schedules of all the users expected to attend the meeting, determines the first time period when all of the people are free (according to the contents of their personal calendars) between 9am and 6pm, and tentatively schedules the meeting. If there is no possible time for the meeting given the constraints, Calendar immediately informs the user who initiated the meeting request; this user can then relax the constraints or negotiate directly with other users to open up some time slots.

If a tentative meeting is scheduled, it appears on all the relevant users' personal calendars in a special section that indicates it is *pending*. Each user must confirm the meeting by entering the meeting into his or her regular appointment section. If some user cannot attend, he or she enters his or her conflicting appointment into the system, and Calendar automatically withdraws the previously scheduled time and tentatively schedules the meeting again in the next available time slot. The meeting remains pending until all users have confirmed for the same time period. The original user can later retract (or cancel) the meeting by deleting the corresponding request from his or her history of commands, which are saved by Calendar.

Figure 1-3 shows two windows, each representing a different user's personal calendar, Josephine on york and Gail on douglass. Figure 1-4 shows what happens after Josephine requests a meeting involving both of them. The automatically selected time appears tentatively on each user's personal calendar. Figure 1-5 shows what happens



```
york-APL > ?
Help:
 type a for some statistics summaries
 type b to begin simulation mode
 type C to clear the list of current events
 type c to show cache information
 type d to dump the raw stats to the terminal
 type D to print the current time and date
 type e to end simulation mode
 type E to print current events
 type f to see file descriptors for hosts which are up
```

```
yor::demo> calendar
```

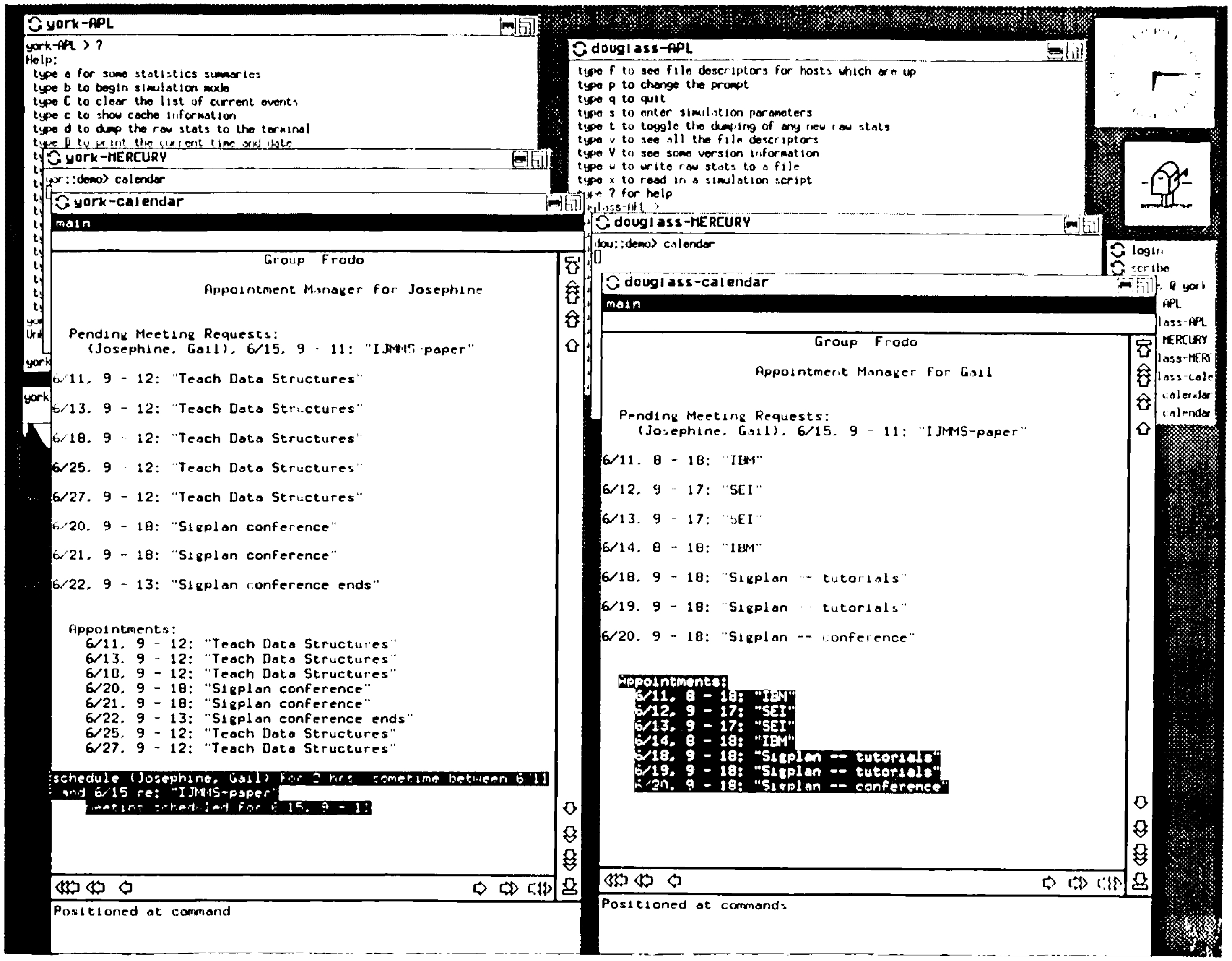
```
main
COMMAND: display-calendar
                    Group  Frodo

          Appointment Manager For Josephine

6/11. 9 - 12: "Teach Data Structures"

6/13. 9 - 12: "Teach Data Structures"

6/18. 9 - 12: "Teach Data Structures"

6/25. 9 - 12: "Teach Data Structures"

6/27. 9 - 12: "Teach Data Structures"

6/20. 9 - 18: "Sigplan conference"

6/21. 9 - 18: "Sigplan conference"

6/22. 9 - 13: "Sigplan conference ends"


  Appointments:
    6/11. 9 - 12: "Teach Data Structures"
    6/13. 9 - 12: "Teach Data Structures"
    6/18. 9 - 12: "Teach Data Structures"
    6/20. 9 - 18: "Sigplan conference"
    6/21. 9 - 18: "Sigplan conference"
    6/22. 9 - 13: "Sigplan conference ends"
    6/25. 9 - 12: "Teach Data Structures"
    6/27. 9 - 12: "Teach Data Structures"

Positioned at commands
```

```
 type f to see file descriptors for hosts which are up
 type p to change the prompt
 type q to quit
 type s to enter simulation parameters
 type t to toggle the dumping of any new raw stats
 type v to see all the file descriptors
 type V to see some version information
 type w to write raw stats to a file
 type x to read in a simulation script
 type ? for help
douglass-APL >
```

```
dou::demo> calendar
```

```
main
COMMAND: display-calendar
                    Group  Frodo

            Appointment Manager For Gail

6/11. 8 - 18: "IBM"

6/12. 9 - 17: "SEI"

6/13. 9 - 17: "SEI"

6/14. 8 - 18: "IBM"

6/18. 9 - 18: "Sigplan -- tutorials"

6/19. 9 - 18: "Sigplan -- tutorials"

6/20. 9 - 18: "Sigplan -- conference"


  Appointments:
    6/11. 8 - 18: "IBM"
    6/12. 9 - 17: "SEI"
    6/13. 9 - 17: "SEI"
    6/14. 8 - 18: "IBM"
    6/18. 9 - 18: "Sigplan -- tutorials"
    6/19. 9 - 18: "Sigplan -- tutorials"
    6/20. 9 - 18: "Sigplan -- conference"

Positioned at commands
```

Figure 1-3: *Two Calendar Schedules before Meeting Request*



```
york-APL > ?
Help:
 type a for some statistics summaries
 type b to begin simulation mode
 type C to clear the list of current events
 type c to show cache information
 type d to dump the raw stats to the terminal
 type D to print the current time and date
```

```
yor::demo> calendar
```

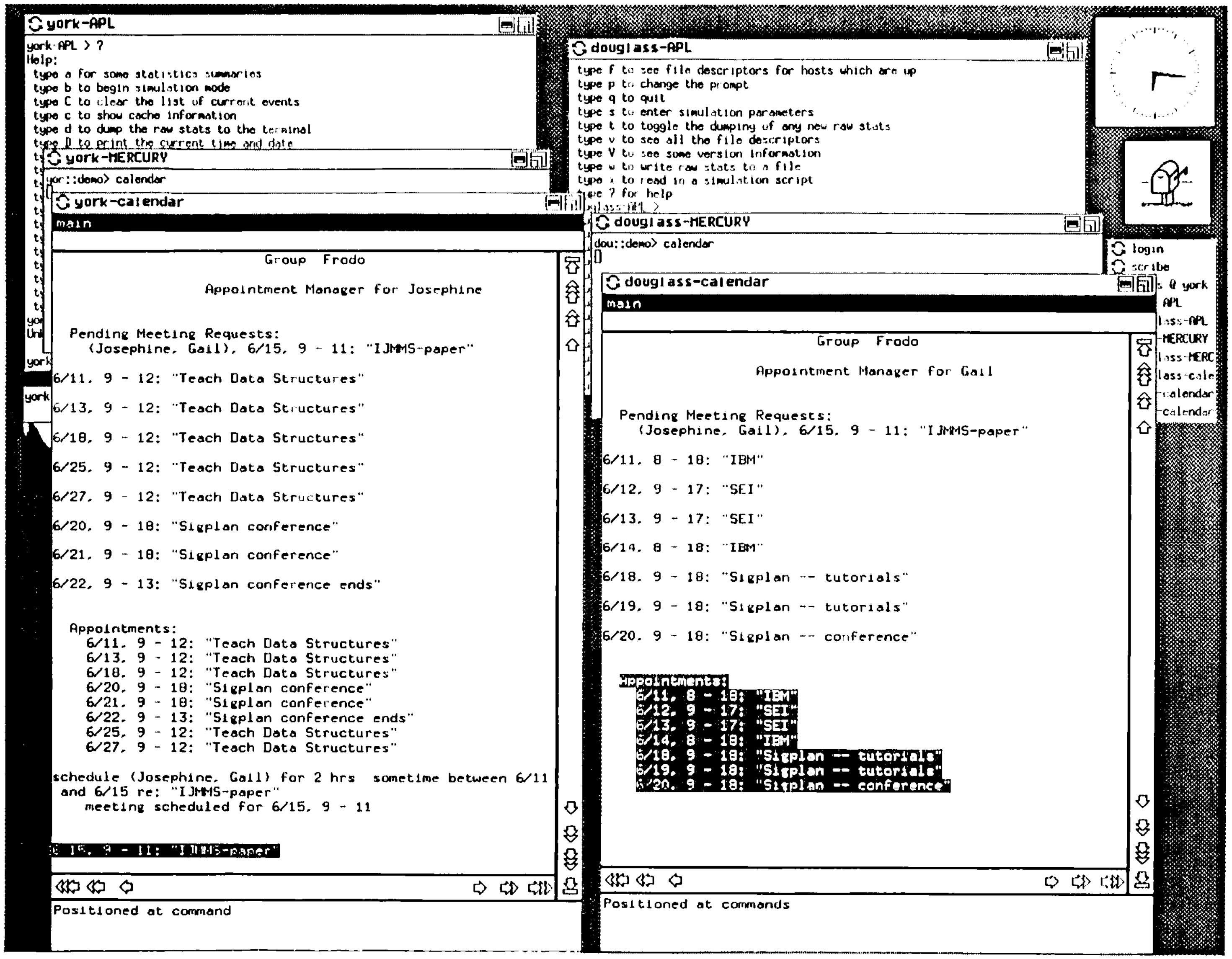
```
                 Group  Frodo

          Appointment Manager for Josephine


 Pending Meeting Requests:
   (Josephine, Gail), 6/15, 9 - 11: "IJMMS-paper"

6/11, 9 - 12: "Teach Data Structures"

6/13, 9 - 12: "Teach Data Structures"

6/18, 9 - 12: "Teach Data Structures"

6/25, 9 - 12: "Teach Data Structures"

6/27, 9 - 12: "Teach Data Structures"

6/20, 9 - 18: "Sigplan conference"

6/21, 9 - 18: "Sigplan conference"

6/22, 9 - 13: "Sigplan conference ends"


 Appointments:
   6/11, 9 - 12: "Teach Data Structures"
   6/13, 9 - 12: "Teach Data Structures"
   6/18, 9 - 12: "Teach Data Structures"
   6/20, 9 - 18: "Sigplan conference"
   6/21, 9 - 18: "Sigplan conference"
   6/22, 9 - 13: "Sigplan conference ends"
   6/25, 9 - 12: "Teach Data Structures"
   6/27, 9 - 12: "Teach Data Structures"

schedule (Josephine, Gail) for 2 hrs  sometime between 6/11
 and 6/15 re: "IJMMS-paper"
   meeting scheduled for 6/15, 9 - 11

Positioned at command
```

```
 type f to see file descriptors for hosts which are up
 type p to change the prompt
 type q to quit
 type s to enter simulation parameters
 type t to toggle the dumping of any new raw stats
 type v to see all the file descriptors
 type V to see some version information
 type w to write raw stats to a file
 type x to read in a simulation script
 type ? for help
```

```
dou::demo> calendar
```

```
                 Group  Frodo

            Appointment Manager for Gail


 Pending Meeting Requests:
   (Josephine, Gail), 6/15, 9 - 11: "IJMMS-paper"

6/11, 8 - 18: "IBM"

6/12, 9 - 17: "SEI"

6/13, 9 - 17: "SEI"

6/14, 8 - 18: "IBM"

6/18, 9 - 18: "Sigplan -- tutorials"

6/19, 9 - 18: "Sigplan -- tutorials"

6/20, 9 - 18: "Sigplan -- conference"


 Appointments:
   6/11, 8 - 18: "IBM"
   6/12, 9 - 17: "SEI"
   6/13, 9 - 17: "SEI"
   6/14, 8 - 18: "IBM"
   6/18, 9 - 18: "Sigplan -- tutorials"
   6/19, 9 - 18: "Sigplan -- tutorials"
   6/20, 9 - 18: "Sigplan -- conference"

Positioned at commands
```

Figure 1-4: *Two Calendar Schedules after Meeting Request*



```
york-APL
york-APL > ?
Help:
 type a for some statistics summaries
 type b to begin simulation mode
 type C to clear the list of current events
 type c to show cache information
 type d to dump the raw stats to the terminal
 type D to print the current time and date
```

```
york-MERCURY
yor::demo> calendar
```

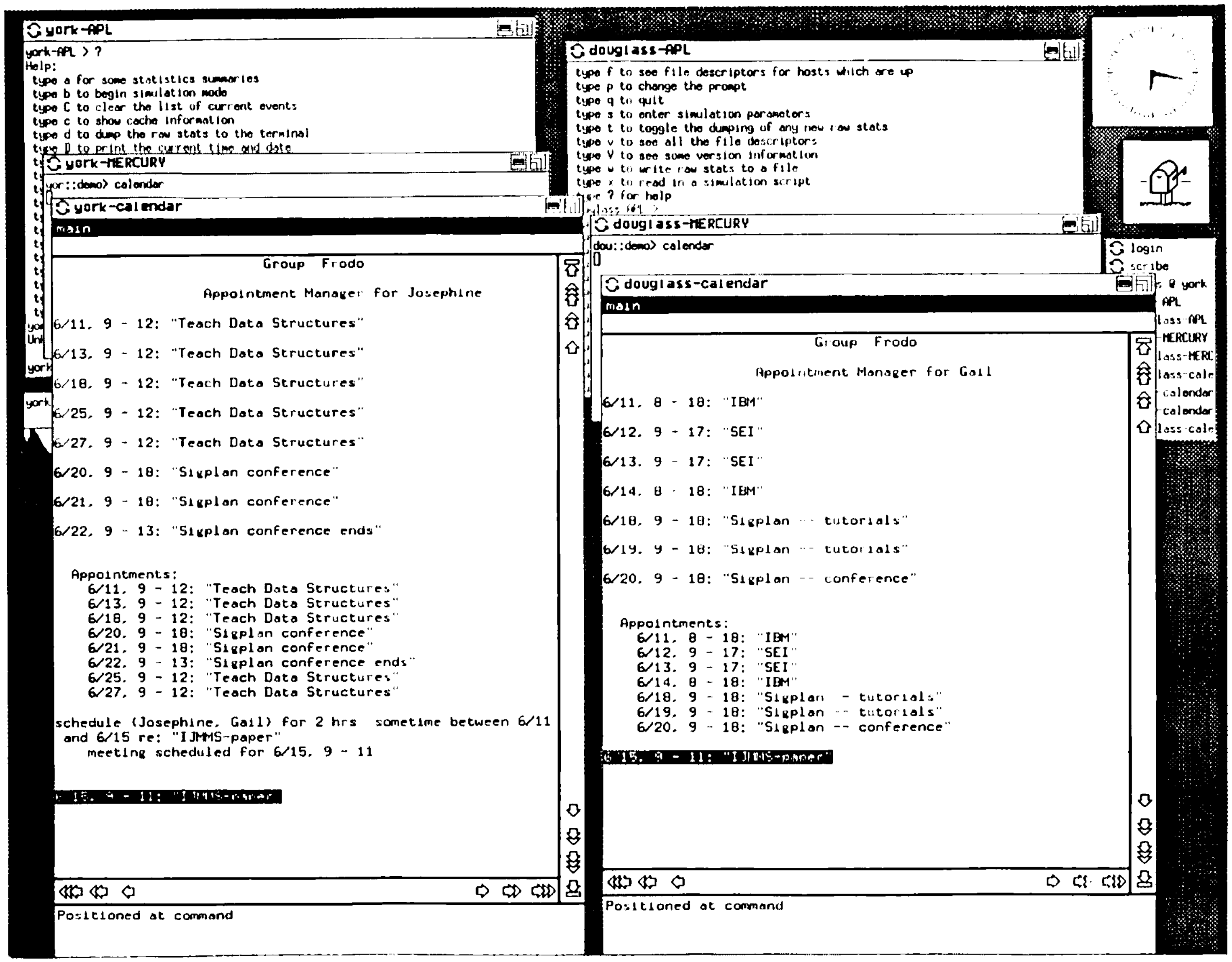
```
york-calendar
main
                    Group  Frodo

              Appointment Manager for Josephine


 Pending Meeting Requests:
   (Josephine, Gail), 6/15, 9 - 11: "IJMMS-paper"

6/11, 9 - 12: "Teach Data Structures"

6/13, 9 - 12: "Teach Data Structures"

6/18, 9 - 12: "Teach Data Structures"

6/25, 9 - 12: "Teach Data Structures"

6/27, 9 - 12: "Teach Data Structures"

6/20, 9 - 18: "Sigplan conference"

6/21, 9 - 18: "Sigplan conference"

6/22, 9 - 13: "Sigplan conference ends"


 Appointments:
   6/11, 9 - 12: "Teach Data Structures"
   6/13, 9 - 12: "Teach Data Structures"
   6/18, 9 - 12: "Teach Data Structures"
   6/20, 9 - 18: "Sigplan conference"
   6/21, 9 - 18: "Sigplan conference"
   6/22, 9 - 13: "Sigplan conference ends"
   6/25, 9 - 12: "Teach Data Structures"
   6/27, 9 - 12: "Teach Data Structures"

schedule (Josephine, Gail) for 2 hrs  sometime between 6/11
 and 6/15 re: "IJMMS-paper"
    meeting scheduled for 6/15, 9 - 11


6/15, 9 - 11: "IJMMS-paper"

Positioned at command
```

```
douglass-APL
type f to see file descriptors for hosts which are up
type p to change the prompt
type q to quit
type s to enter simulation parameters
type t to toggle the dumping of any new raw stats
type v to see all the file descriptors
type V to see some version information
type w to write raw stats to a file
type x to read in a simulation script
type ? for help
douglass-APL >
```

```
douglass-MERCURY
dou::demo> calendar
```

```
douglass-calendar
main
                    Group  Frodo

              Appointment Manager for Gail


 Pending Meeting Requests:
   (Josephine, Gail), 6/15, 9 - 11: "IJMMS-paper"

6/11, 8 - 18: "IBM"

6/12, 9 - 17: "SEI"

6/13, 9 - 17: "SEI"

6/14, 8 - 18: "IBM"

6/18, 9 - 18: "Sigplan -- tutorials"

6/19, 9 - 18: "Sigplan -- tutorials"

6/20, 9 - 18: "Sigplan -- conference"


 Appointments:
   6/11, 8 - 18: "IBM"
   6/12, 9 - 17: "SEI"
   6/13, 9 - 17: "SEI"
   6/14, 8 - 18: "IBM"
   6/18, 9 - 18: "Sigplan -- tutorials"
   6/19, 9 - 18: "Sigplan -- tutorials"
   6/20, 9 - 18: "Sigplan -- conference"

Positioned at commands
```

Figure 1-5: *Two Calendar Schedules after Original User Confirms*



```
york-APL > ?
Help:
 type a for some statistics summaries
 type b to begin simulation mode
 type C to clear the list of current events
 type c to show cache information
 type d to dump the raw stats to the terminal
 type D to print the current time and date
```

```
yor::demo> calendar
```

```
                 Group  Frodo

         Appointment Manager for Josephine

6/11, 9 - 12: "Teach Data Structures"

6/13, 9 - 12: "Teach Data Structures"

6/18, 9 - 12: "Teach Data Structures"

6/25, 9 - 12: "Teach Data Structures"

6/27, 9 - 12: "Teach Data Structures"

6/20, 9 - 18: "Sigplan conference"

6/21, 9 - 18: "Sigplan conference"

6/22, 9 - 13: "Sigplan conference ends"


  Appointments:
    6/11, 9 - 12: "Teach Data Structures"
    6/13, 9 - 12: "Teach Data Structures"
    6/18, 9 - 12: "Teach Data Structures"
    6/20, 9 - 18: "Sigplan conference"
    6/21, 9 - 18: "Sigplan conference"
    6/22, 9 - 13: "Sigplan conference ends"
    6/25, 9 - 12: "Teach Data Structures"
    6/27, 9 - 12: "Teach Data Structures"

schedule (Josephine, Gail) for 2 hrs  sometime between 6/11
 and 6/15 re: "IJMMS-paper"
    meeting scheduled for 6/15, 9 - 11


6/15, 9 - 11: "IJMMS-paper"

Positioned at command
```

```
type f to see file descriptors for hosts which are up
type p to change the prompt
type q to quit
type s to enter simulation parameters
type t to toggle the dumping of any new raw stats
type v to see all the file descriptors
type V to see some version information
type w to write raw stats to a file
type x to read in a simulation script
type ? for help
```

```
dou::demo> calendar
```

```
                 Group  Frodo

            Appointment Manager for Gail

6/11, 8 - 18: "IBM"

6/12, 9 - 17: "SEI"

6/13, 9 - 17: "SEI"

6/14, 8 - 18: "IBM"

6/18, 9 - 18: "Sigplan -- tutorials"

6/19, 9 - 18: "Sigplan -- tutorials"

6/20, 9 - 18: "Sigplan -- conference"


  Appointments:
    6/11, 8 - 18: "IBM"
    6/12, 9 - 17: "SEI"
    6/13, 9 - 17: "SEI"
    6/14, 8 - 18: "IBM"
    6/18, 9 - 18: "Sigplan -- tutorials"
    6/19, 9 - 18: "Sigplan -- tutorials"
    6/20, 9 - 18: "Sigplan -- conference"

6/15, 9 - 11: "IJMMS-paper"

Positioned at command
```

Figure 1-6: *Two Calendar Schedules after Other User Confirms*

when Josephine confirms the meeting. Note that Gail has not yet confirmed, and the meeting time is still regarded as pending. Figure 1-6 shows what happens after Gail confirms.

The full attribute grammar specification of the calendar application is given in appendix B. The basic MERCURY facilities provide all the support needed to propagate information among multiple users whose calendars reside on their personal workstations. A prototype of an application can be implemented very quickly[1] by specifying the structure of the document involved (the set of personal calendars in this case), the semantic dependencies among the components of the document segments (*e.g.*, the constraints for scheduling meetings among busy users), and the semantic dependencies within each component (*e.g.*, multiple meetings involving the same user cannot be overlapping).

1.4. Overview of Technical Results

This thesis describes a framework based on the attribute grammar formalism, and supporting algorithms, necessary to make multi-user semantics-based editors work for both programming and non-programming computer-supported cooperative work scenarios, such as those described. In particular, we present algorithms that propagate changes among the multiple users of an application and update any other components of the application dependent on the changed component(s).

1.4.1. Multiple Asynchronous Modifications

The dependencies among multiple components of the program (or other structured document)[2] and the computations that take place when components are changed — to update their dependent components accordingly — are expressed in an attribute

[1]The initial implementation of the Calendar functionality took only two days, although more time was spent working on the user interface.

[2]Throughout the rest of this thesis, the term ''program'' should be understood to mean any kind of structured document segmented among multiple users according to the requirements of the application, such as the distributed calendar above.

grammar (AG), extended as described in subsection 1.4.2 to support multi-module programs. We consider only the general class of *noncircular* AGs and its subclasses, where an attribute of a component cannot depend (even indirectly) on itself. The program is represented internally as an attributed derivation tree. Each entry or change to the program is treated as the *replacement* of a subtree in the derivation tree. When there are multiple users, the subtree replacements are *asynchronous*.

After a subtree replacement occurs in the attributed derivation tree, the values of attributes in the unchanged part of the tree may be *inconsistent* with respect to the new subtree, and similarly, attributes in the new subtree may be inconsistent with respect to the rest of the tree. One way to reestablish attribute consistency in the derivation tree is by an exhaustive recomputation of all the attributes in the entire tree after every change. However, this would result in unacceptable response time for the users, especially for the multi-user case, and would often do much more work than necessary. Instead, *incremental evaluation* is employed, whose goal is to compute only those attributes actually affected by the change.

An optimal incremental evaluation algorithm for single subtree replacements, applicable to arbitrary noncircular AGs, was developed by Reps, Teitelbaum and Demers [Reps 83]. Their algorithm employs a scheduling graph, called a *model*, to keep track of attributes that need to be evaluated, and direct and transitive dependency edges among them. The model orders the evaluation of attributes to ensure that an attribute is evaluated only after all the attributes it depends on have been assigned their final values (*i.e.*, final until the next subtree replacement). The algorithm performs the minimal amount of work necessary to reestablish attribute consistency in the derivation tree following a single subtree replacement. That is, both the number of attribute evaluations and the bookkeeping costs of the algorithm are proportional to the size of *AFFECTED*, the set of attributes in the updated tree whose values changed.

A naive approach to handling *k* asynchronous subtree replacements is to perform *k* sequential applications of the single-edit algorithm of Reps *et al.*, where new changes and thus new evaluation processes are blocked until the previous one has completed. However, if the sets of attributes affected by multiple subtree replacements are not disjoint, this approach would always evaluate attributes in the intersection more than once, even though information might have been available to prevent unnecessary evaluations. In contrast to the naive approach, the algorithms for multiple asynchronous subtree replacements presented in this thesis initiate an evaluation process immediately following a subtree replacement, and ''combine'' the evaluation processes of multiple subtree replacements to avoid evaluating attributes in the intersection more than once.

Consider k asynchronous edits whose corresponding evaluation processes have been initiated but have not yet terminated. We define the set *ASYNC–AFFECTED* to be the minimal set of attributes that must be evaluated to restore consistency in the attributed derivation tree modified by the k edits. The size of *ASYNC–AFFECTED* depends on the timing of the subtree replacements. When the k subtree replacements occur ''almost sequentially'', that is, the attributes affected by the i^{th} subtree replacement have almost all been evaluated before the $(i + 1)^{\text{st}}$ subtree replacement occurs, then

$$|ASYNC\text{–}AFFECTED_{SEQ}| = \sum_{i=1}^{k} |AFFECTED_i|$$

where $AFFECTED_i$ is the set of attributes affected by the i^{th} subtree replacement. In this case, attributes affected by j subtree replacements, $j \leq k$, are necessarily evaluated j times.

When the k subtree replacements occur ''almost simultaneously'', *ASYNC–AFFECTED* contains the set of attributes with different values in the two trees T and T^k, where T is the consistently attributed derivation tree before the k subtree replacements occurred, and T^k is the updated tree after the evaluation of all k subtree replacements has completed. The size of *ASYNC–AFFECTED* for k ''almost simultaneous'' edits may be less than the cardinality of the union of the k affected sets, since an attribute's value may have been changed as a result of one edit, only to be changed back to its original value because of a subsequent edit. That is,

$$|ASYNC\text{–}AFFECTED_{SIM}| \leq |UNION\text{–}AFFECTED|, \text{ where}$$
$$UNION\text{–}AFFECTED = AFFECTED_1 \cup AFFECTED_2 \cup \cdots \cup AFFECTED_k.$$

When the timing of the k subtree replacements is in between these two boundary cases, reestablishing attribute consistency in the modified tree may require additional attribute evaluations than if the k edits were ''almost sequential'', but may avoid some attribute evaluations that would have been performed had the the edits been ''almost simultaneous''. That is, the size of the set *ASYNC–AFFECTED* is

$$|ASYNC\text{–}AFFECTED_{SIM}| \leq |ASYNC\text{–}AFFECTED| \leq |ASYNC\text{–}AFFECTED_{SEQ}|$$

We state that an incremental attribute evaluation algorithm for multiple asynchronous subtree replacements is *semi-optimal* if the number of attribute evaluations performed to reestablish attribute consistency is proportional to the size of *ASYNC–AFFECTED.* In other words, for any $k > 1$ modifications affecting the same attribute a, where the k evaluation processes are still in progress and none have yet evaluated a, the algorithm evaluates a at most once. The algorithm is *optimal* if the total work performed to update the tree after k asynchronous edits is proportional to the size of *ASYNC–AFFECTED.*

We have developed a number of semi-optimal incremental evaluation algorithms for multiple asynchronous subtree replacements, which are applicable to different classes of attribute grammars. We do not have an optimal solution, however, and it remains an open question whether an optimal algorithm for multiple asynchronous edits can be designed.

Our *Collision-Merging* algorithm, applicable to arbitrary noncircular AGs, allows the k evaluation processes corresponding to the k subtree replacements to proceed independently while they cover disjoint parts of the derivation tree. A model is maintained for each independent evaluation process in progress. As a model expands to include attributes dependent on other attributes that have changed in value, it may expand to cover a derivation tree node that is already included in <u>another</u> model arising from a different subtree replacement that is being evaluated concurrently. When this happens, the two models are *merged* into one, thereby combining the corresponding evaluation processes that had been proceeding independently previously, and evaluation continues with the merged model.

This merging is useful because an attribute that appears in multiple models may be scheduled and thus evaluated multiple times. If all the models in which it appears are merged, however, then it will be evaluated at most once, as desired. This optimization of course requires that the attribute is still in both models at the time of the merge. If an attribute was previously in one model, but scheduled, evaluated and removed before the two models were merged, then it will have to be evaluated a second time for the second model. However, if there are some other attributes still in the first model that depend on the removed attribute, they will be evaluated only once.

Our solution for arbitrary noncircular grammars attempts to find a balance between the number of attributes evaluated and the bookkeeping overhead of the algorithm. Separate evaluation processes, represented by their models, are merged as soon as they cover the same node in the derivation tree, and thus most unnecessary attribute evaluations are avoided. However, this algorithm may evaluate attributes that are not in *ASYNC–AFFECTED* (although it will never evaluate attributes that are not in $AFFECTED_i$, for $1 \leq i \leq k$). The bookkeeping costs of the algorithm for k asynchronous subtree replacements are $O(\sum_{i=1}^{k} |AFFECTED_i| \cdot i)$ in the worst case.

The Collision-Merging algorithm uses a *dynamic* evaluation strategy, that is, it employs scheduling graphs that are computed at run-time from the dependencies in the attributed tree. *Static* evaluation algorithms, in contrast, use precomputed *plans* that specify the order of evaluation of attributes for each production in the grammar. Static evaluators are more efficient, in both time and space, than dynamic evaluators for both incremental and exhaustive attribute evaluation, but they can only be constructed for subclasses of the noncircular attribute grammars.

We have also developed a new static incremental attribute evaluation algorithm for multiple asynchronous edits, applicable to ordered attribute grammars (OAGs). OAGs, originally defined by Kastens [Kastens 80], form a proper subclass of noncircular AGs. Our algorithm for OAGs minimizes the number of attributes evaluated by interleaving the plans activated by the k subtree replacements. The overhead of the algorithm results from the cost of scheduling plans activated by different subtree replacements according to which one should be executed first. The worst-case bookkeeping costs of this algorithm for k subtree replacements is $O(|ASYNC\text{–}AFFECTED| \cdot n \cdot k)$, where n is the size of the derivation tree.

The factor n in the cost of the OAG evaluator arises from traversing the derivation tree to determine the interleaving order of plans associated with nodes in different parts of the tree. In practice, this is not as bad as it seems — experiments using the GAG system, an AG-based compiler generator, showed that the total amount of time required by the evaluator to move from node to node is insignificant compared with the time needed for attribute evaluation [Kastens 82]. Furthermore, the conditions causing the worst-case behavior of the algorithm is pathological and not expected to arise in practice.

We define a subclass of OAGs, called the *pairwise ordered attribute grammars*, where the interleaving order for each pair of plans in the grammar is always the same, independent of the derivation tree. Thus, a table can be constructed from an analysis of the grammar containing the scheduling order for plans of the grammar, and a table-lookup operation replaces the tree traversal needed for OAGs during attribute evaluation. The overhead of the evaluation algorithm for the pairwise ordered AGs is reduced to $O(|ASYNC{-}AFFECTED| \cdot \log n \cdot k)$.

We now outline our results for the second problem addressed in this thesis — the extension of the attribute grammar formalism to support segmentation of the program into modular units.

1.4.2. Segmentation of Derivation Tree

Prior to this work, the attribute grammar formalism was primarily used to specify the static semantics of languages for programming-in-the-small, where the entire program is contained in a single file. Attribute grammars for languages with programming-in-the-large facilities define only the intra-module semantics since the formalism cannot directly express inter-module semantics. We *extend* the attribute grammar formalism to express programming-in-the-large constructs found in real programming languages, including textual inclusion, multiple kinds of compilation units, and nested compilation units, thus unifying inter-module and intra-module static semantics.

We define *segmentable context-free grammars*, the underlying substrate of extended attribute grammars. Certain nonterminal symbols in segmentable context-free grammars are declared to be *distributable*, indicating that they derive a separate modular unit, generically called a segment. A segment may interface to exactly one other segment (to represent nested program units) or to several other segments (to represent textually included units).

Productions in a segmentable context-free grammar can derive an unordered collection of segments. Standard context-free grammars can define only ordered collections (lists) by a skewed tree using a left- or right-recursive pair of context-free productions. The ability to express an unordered collection of segments is necessary to eliminate the cost of synchronizing the addition of new segments to the program by multiple users.

We define the class of *segmentable attribute grammars* by extending the definitions of attributes, and the built-in operators available for use in their semantic equations, to employ segmentable context-free grammars. We describe how existing attribute evaluation algorithms can be extended to *local segment evaluators* for attribute evaluation within segments, and combined with a *global evaluator* for intersegment linkage and propagation of attribute values across interface nodes. Such a segmented attribute evaluator can be used for detecting interface errors of a segmented derivation tree.

We identify a particular anomaly that arises when attributes are propagated from one segment into another and then back to the first segment, potentially causing the first segment to remain inconsistently attributed for a considerable amount of time when the segments reside on different machines. We describe a technique for *summarizing* attributes in such a way that evaluation is not delayed in the first segment due to the propagation through the second segment. Our summarizing technique involves the transformation of an attribute grammar into an equivalent one, and is only applicable to a subclass of the segmentable attribute grammars.

Our result is a uniform approach to formal specification of both intra-module and inter-module static semantic properties, that is, both within and between segments, with the ability to use attribute evaluation algorithms to carry out a complete static semantic analysis of a multi-module program.

1.5. Related Work

Reps, Teitelbaum and Demers [Reps 83] developed the first optimal algorithm for incremental evaluation of attribute grammars, optimal in the sense that the complexity is proportional to the number of attributes whose value is changed by the subtree replacement, $O(|AFFECTED|)$. Their algorithm allows only a single editing cursor (*i.e.*, a user-controlled pointer to the displayed text that represents a derivation tree node where a subtree replacement can occur), with all cursor movements restricted to be from a child node to its parent or *vice versa*, and was intended to support single-user semantics-based editors. The user makes a subtree replacement, ''waits'' during the subsequent evaluation process (the delay is generally unnoticeable since the program is small), and then can move the cursor and make another change.

An algorithm for multiple synchronous subtree replacements was developed by Reps, Marceau and Teitelbaum [Reps 86]. Synchronous changes support commands, such as program transformations, that do not map nicely to single subtree replacements. The idea is that all the subtree replacements are made first and collected — and then a single process is employed for overall incremental evaluation. This algorithm is not effective when changes occur asynchronously, and thus is not suitable for multiple users.

Kaplan and Kaiser were the first to describe a (distributed) attribute evaluation algorithm to handle multiple asynchronous edits on program modules that are distributed across a number of workstations connected by a local area network [Kaplan 86]. Their algorithm, for arbitrary noncircular AGs, was later expanded for parallel evaluation on a centralized or decentralized tree in response to multiple asynchronous edits [Kaiser 90]. Their initial work left open the problems addressed in this thesis. Their updated algorithms assume the merging and other support algorithms described in this thesis.

A number of other algorithms for handling multiple asynchronous subtree replacements have been developed in the same time frame as the work reported in this thesis, but all except one of the others are applicable only to semantic dependencies that are expressible in some restricted subclass of the noncircular attribute grammars — while we have designed algorithms for both the general class and subclasses of it. The restrictions may not be a problem for semantics-based editors for programming, since many programming languages fit within the restrictions, but the requirements of other application domains may be different.

Geitz [Geitz 87] describes an algorithm that minimizes the number of attributes evaluated by maintaining transitive dependency edges between attributes in one model that depend on attributes in another model. This algorithm is applicable only to a subset of what is called the partitioned attribute grammars, for which the transitive dependency information required by the algorithm can be computed statically from the AG. (The algorithm for multiple synchronous changes by Reps, Marceau and Teitelbaum cited above also works only for this class).

Peckham [Peckham 90] developed static incremental evaluators for multiple <u>synchronous</u> or <u>asynchronous</u> subtree replacements, which are applicable to a subset of the partitioned AGs called the globally partitionable attribute grammars. Peckham's algorithms evaluate the minimal number of attributes, with worst-case bookkeeping costs of $O(|ASYNC\text{-}AFFECTED| \cdot \log n \cdot k)$ and $O(|ASYNC\text{-}AFFECTED| \cdot n \cdot k)$ for the synchronous and asynchronous versions, respectively.

Hoover [Hoover 87] is primarily concerned with incremental graph evaluation, to support modifications of dependency graphs and evaluation of what he calls *influenced* vertices of the graph (influenced vertices correspond approximately to affected attributes). He discusses incremental attribute evaluation for arbitrary noncircular attribute grammars in this framework, which also applies to a range of other incremental computation problems. Most of his results are based on the paradigm where modification and evaluation alternate, and both are done sequentially. However, Hoover relaxes this assumption and briefly addresses the problems of allowing dependency graph updates while evaluation is in progress and of parallel evaluation on a dependency graph. Hoover's solution to these problems employs an approximate rather than exact topological order, and therefore it is possible that his algorithm will unnecessarily evaluate the same attribute multiple times even for a single dependency graph modification.

Boehm and Zwaenepoel describe a non-incremental distributed/parallel evaluation technique for use in compilers that divides the derivation tree into non-nested bottom subtrees and a remaining top tree [Boehm 87]. In their implementation, these subtrees are evaluated on different workstations connected by a high-speed network using a combined static/dynamic evaluation strategy. The bottom subtrees are evaluated entirely statically using the ordered attribute evaluation strategy, while the top tree is evaluated dynamically by a topological sort of its dependency graph. The combination of these evaluation strategies arose from their observation that dynamic evaluators are easy to parallelize but require a lot of storage while static evaluators are inherently sequential but utilize memory efficiently. Boehm and Zwaenepoel's work can be used to speed up the compilation of small programs or individual modules of a large program, but must be combined with our segmentable attribute grammars to compile (with inter-module static semantic analysis) programs with multiple modules.

Alblas describes a parallel incremental evaluator that is an incremental version of the combined static/dynamic evaluator of Boehm and Zwaenepoel [Alblas 90]. The goal of his work is to support asynchronous subtree replacements caused by transformations in different regions of the derivation tree. The application of a particular transformation depends on attribute values in the tree. Alblas' evaluation algorithm allows the tree to have inconsistent attributes as long as "safety criteria" are met. The safety criteria ensure that a transformation of an inconsistent tree is enabled only if it would have been enabled had the tree been consistent. Thus, calling the evaluator is delayed when determined to be safe, allowing the evaluation processes of several transformations to proceed concurrently. An evaluation algorithm that allows inconsistent attributes is inappropriate for semantics-based editors, since the attribute values inform the user about the program's consistency.

All of the above efforts are concerned with semantics-based editors for programming, and have not addressed other applications. However, Hudson and King have applied incremental attribute evaluation to user interface updates [Hudson 88] and more general database applications [Hudson 89] in their Cactis system. They do not support multiple users or asynchronous subtree replacements. We anticipate that combining their results with ours will dramatically expand the range of opportunities for computer-supported cooperative work.

1.6. Organization of Thesis

The thesis is organized as follows. We give a brief introduction to attribute grammars in section 2. In this chapter we also describe in detail the optimal attribute evaluation algorithm due to Reps, Demers, and Teitelbaum [Reps 83] for single subtree replacements, which is the basis for our general algorithm for multiple asynchronous subtree replacements for noncircular AGs.

The problem of incremental attribute evaluation for multiple asynchronous subtree replacements is the topic of chapters 3 and 4. The evaluation strategy of chapter 3 is dynamic, while that of chapter 4 is static.

The algorithms described in chapter 3 are applicable to the general class of noncircular attribute grammars. In this chapter, we are concerned with two issues. First, we discuss how to correctly maintain the underlying data structures used by the evaluation algorithm to determine the dependencies among attributes of the derivation tree, since an edit may change the (indirect) dependencies among attributes affected by other asynchronous edits. Second, we present algorithms to merge the scheduling graphs used by each attribute evaluation process in progress, to avoid unnecessary repeated evaluations of the same attribute.

A new incremental evaluation algorithm for ordered attribute grammars that minimizes the number of attributes reevaluated when there are asynchronous program modifications is described in chapter 4. We define pairwise ordered attribute grammars, a subclass of OAGs for which the scheduling information necessary for asynchronous subtree replacements can be precomputed during construction of the evaluator.

Chapters 5, 6, and 7 deal with the problem of extending the attribute grammar formalism to handle separate modular units. Segmentable context-free grammars are introduced in chapter 5. Using these grammars, one can represent multi-module programs by segmented derivation trees. Such a representation is essential for multi-user semantics-based editors, where the program being developed is decomposed into modules and assigned to individual members of the development team.

Segmentable context-free grammars provide the underlying substrate for segmentable attribute grammars, which are the topic of chapter 6. Besides defining this class of attribute grammars, in this chapter we also discuss the issues of attribute evaluation in the context of programming-in-the-large, particularly the separation of concerns between the local evaluator for each segment and the global evaluator that propagates attributes across segment boundaries.

Chapter 7 defines the conditions that cause a delay in evaluating attributes in one segment because of attribute dependencies through a different segment. We describe how some segmentable AGs can be transformed to avoid this situation.

We conclude in chapter 8 by summarizing the contributions of the research reported in this thesis. We list a number of problems that are natural extensions of this work, and also discuss some interesting open problems for future work.

In appendix A we describe the MERCURY system, which generates multi-user semantics-based editors from attribute grammar specifications. In our prototype implementation, a program (or other structured document) consists of a collection of segments; that is, MERCURY supports only a flat segment organization. The attribute evaluation algorithm used in MERCURY for multiple asynchronous subtree replacements is the one described in chapter 3. The attribute grammar specification for the Calendar application is given in appendix B.

Chapter 2
Background

2.1. Attribute Grammars

Attribute grammars were first introduced by Knuth to describe the context-sensitive properties (static semantics) of programming languages [Knuth 68]. An attribute grammar (AG) is based on a context-free grammar (CFG) that describes the language's syntax. A context-free grammar is denoted as $G = (N, T, P, S)$, where N and T are finite sets of nonterminal and terminal symbols respectively, P is a finite set of productions, and S is the start symbol of the grammar. Productions in a context-free grammar are of the form $X \rightarrow \alpha$, where X is a nonterminal and α is a string of symbols from $(N \cup T)^*$.

An AG extends a context-free grammar G by associating a set $A(X)$ of *attributes* with each symbol X in G. Each attribute represents a specific property of the symbol, and can take on any of a specified set of values. The notation $X.a$ indicates that attribute a is an element of $A(X)$. *Semantic equations* defining these attributes are associated with productions of the grammar G. A semantic equation defines an attribute, a_0, as the value of a *semantic function* applied to other attributes of that production, $a_1, \ldots, a_k$. The attribute on the left-hand side of the equation, a_0, is *functionally dependent* on the attributes on the right-hand side, $a_1, \ldots, a_k$.

Attributes are divided into two disjoint classes: *inherited* and *synthesized*. The inherited and synthesized attributes of a symbol X are denoted by $I(X)$ and $S(X)$ respectively; $I(X) \cup S(X) = A(X)$ and $I(X) \cap S(X) = \emptyset$. A semantic equation defines a synthesized attribute of the left-hand symbol of a production, or an inherited attribute of one of the right-hand side symbols. The start symbol of the grammar, S, has no inherited attributes; that is, $I(S) = \varnothing$. In Knuth's original formulation of AGs, terminal symbols could have inherited attributes but no synthesized ones. We follow the approach of later work on AGs and make no distinction between terminal and nonterminal symbols.

The *output attributes* of a production p: $X_0 \rightarrow X_1 \cdots X_n$ are those attributes defined by semantic equations associated with p. These are the synthesized attributes of the left-hand side symbol of p (*i.e.*, $S(X_0)$) and the inherited attributes of the right-hand side symbols of p (*i.e.*, $I(X_1) \cup \cdots \cup I(X_n)$). The *input attributes* of p are those attributes which appear on the right hand side of the semantic equations of p.

An AG is in *Bochmann normal form* if for any production p, the input attributes of p consist of the inherited attributes of the left-hand symbol of p (*i.e.*, $I(X_0)$), and the synthesized attributes of the right-hand side symbols of p (*i.e.*, $S(X_1) \cup \cdots \cup S(X_n)$) [Bochmann 76]; that is, no attribute defined in p can be used to define another attribute in p. In this thesis, we assume that attribute grammars are in Bochmann normal form. Any attribute grammar can be converted to Bochmann normal form [Bochmann 76], so this is not a limiting assumption.[3]

Figure 2-1 gives an example of an attribute grammar fragment for declarations in a Pascal-like programming language. There are four productions in the context-free grammar, *p1* through *p4*. Each symbol in a production has associated attribute instances (declared in figure 2-1 (a)), and each production has associated semantic equations that define the values of the attribute instances (shown in figure 2-1 (b)). Occurrences of the same symbol within one production are distinguished by the use of a numerical suffix; for example, in production *p3*, there are two occurrences of *Decls*, denoted by *Decls$1* and *Decls$2* for the first and second occurrence, respectively.

The AG of figure 2-1 builds a symbol table for all declared identifiers, and also marks identifiers that are declared more than once as erroneous. The functions *Member* and

[3]This conversion may require duplication of semantic functions. For example, if the semantic equations associated with the production p: $X_0 \rightarrow X_1 X_2$ are $X_1.a = f(X_0.a), X_2.a = X_1.a$, the equivalent normal form version would require two invocations of the semantic function f, resulting in the semantic equations $X_1.a = f(X_0.a), X_2.a = f(X_0.a)$.

```
Decls:  { synthesized attributes:    SymTabOut;
          inherited attributes:      SymTabIn; }

Decl:   { synthesized attributes:    SymTabOut, error;
          inherited attributes:      SymTabIn; }

Id:     { synthesized attributes:    Name;
          inherited attributes:      ∅; }

Type:   { synthesized attributes:    TpKind;
          inherited attributes:      ∅; }
```

(a): *Context-free Symbols of the Attribute Grammar and their Attributes*

```
p1:  Program ::= · · · Decls · · ·
          { Decls.SymTabIn = NullTbl(); }

p2:  Decls ::= /* empty rule */
          { Decls.SymTabOut = Decls.SymTabIn; }

p3:  Decls$1 ::= Decl Decls$2
          { Decl.SymTabIn = Decls$1.SymTabIn;
            Decls$2.SymTabIn = Decl.SymTabOut;
            Decls$1.SymTabOut = Decls$2.SymTabOut; }

p4:  Decl ::= Id ':' Type ';'
          { Decl.error = Member(Decl.SymTabIn, Id.Name)
                          ? "<-- Variable already declared"
                          : "";
            Decl.SymTabOut = Insert(Decl.SymTabIn, Id.Name, Type.TpKind); }
```

(b): *Productions of the Attribute Grammar Fragment and their Semantic Equations*

Figure 2-1: *An Attribute Grammar Example*

Insert, used in the semantic equations associated with production *p4*, are defined as part of the AG specifications; the definitions of these functions are omitted from figure 2-1. AG functions are pure functions, that is, they have no side effects. Their only arguments are constants or other attribute occurrences of the production. The expression language for writing semantic equations used in this thesis is self-explanatory except where noted. It includes conditional expressions written with the ternary operator "? :"; see, for example, the semantic equation defining *Decl.error* in production *p4*.

The value of an attribute instance is computed according to its defining semantic equation. Before an attribute can be evaluated, all other attributes that it is functionally dependent on must have already received values. The functional dependencies among the attributes in the tree create a partial ordering on the attribute instances in the tree. Any attribute evaluation algorithm must obey this partial order, but since the ordering is partial, there may be more than one order of evaluating the attribute instances of the tree.

We illustrate these concepts by an example. Consider the following string derived from the CFG of figure 2-1:

```
    a: integer;
    b: boolean;
```

Figure 2-2 (a) shows the derivation tree for this string;[4] the nodes in the tree are labeled with symbols of the context-free grammar. Figure 2-2 (b) is the semantic tree for the same declarations.[5] A *semantic tree* is a derivation tree where each tree node additionally contains fields corresponding to the attributes of its labeling grammar symbol. The *dependency graph* of a semantic tree *T*, denoted by *D(T)*, represents functional dependencies among the attribute instances of *T*, and is defined as follows: *D(T)* is a directed graph, (V, E), where

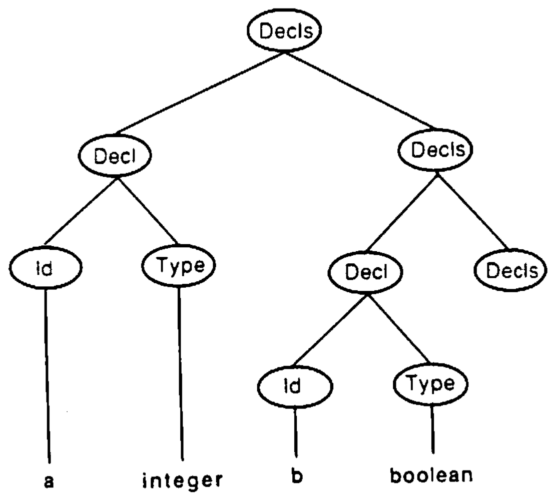
- $V = \{$ attribute instances of T $\}$, and
- $E = \{ (a,b) \mid a, b \in V$, and a is an argument of $b \}$.

The dependency graph for our running example is shown in figure 2-2 (c).

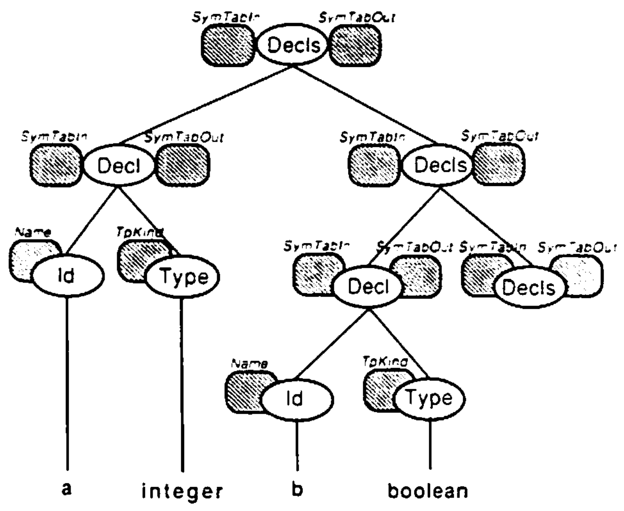
Knuth describes a simple algorithm for evaluating all the attributes in a semantic tree [Knuth 68]. The algorithm makes use of the dependency graph of the semantic tree. The vertices of the dependency graph, which correspond to attribute instances in the

[4]We are imprecise here, since this string cannot be matched from the start symbol of the grammar, but it should be clear what is meant.

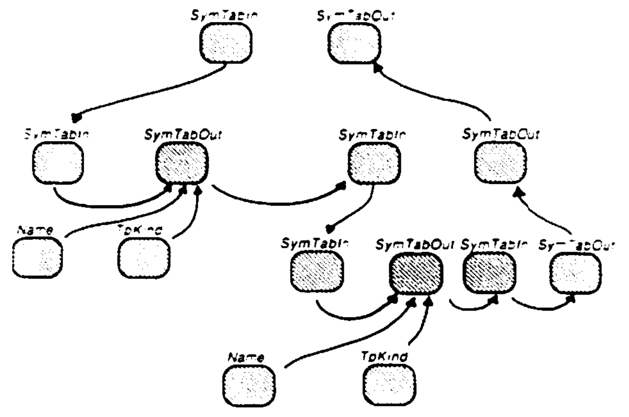
[5]The *error* attribute is not shown in the figure for clarity.



(a) *A Derivation Tree*



(b) *A Semantic Tree*



(c) *A Dependency Graph*

LEGEND

⬭	*derivation tree node: nonterminal occurrence*
▨	*attribute field in semantic tree node: attribute instance*

Figure 2-2: *A String Derived from the Grammar of Figure 2-1*

semantic tree, are first topologically sorted. Then, the attribute instances are evaluated according to their topological order. This algorithm only works if the dependency graph is acyclic [Knuth 71]. Many attribute evaluators assume that the AG is *noncircular*, that is, that the dependency graph for any string derived by the grammar is acyclic. However, it is hard to verify this assumption since to do so requires exponential time [Jazayeri 75]. In this thesis, we only consider noncircular AGs. (The class of noncircular attribute grammars is also referred to as the *well-defined* AGs.)

Although the algorithm described above is simple and works for any noncircular AG, its performance is poor for both time and space: all decisions are made at run-time, and the dependency graph must be built and stored. More efficient algorithms for performing attribute evaluation have been developed for use in practical compiler-compilers [Farrow 84, Kastens 82, Ganzinger 77]. These algorithms perform most of the work at grammar-analysis time, once for each AG, thus improving the performance of the evaluator. The disadvantage of these evaluators, which are called *static* or *semi-static* evaluators depending on how much work is done at run-time, is that they do not work for all noncircular AGs but only for subclasses of them.

2.2. Incremental Attribute Evaluation

The use of attribute grammars in semantics-based editors was originated by Demers, Reps and Teitelbaum [Demers 81]. A program is represented internally by its semantic tree. The program is modified by a sequence of pruning and grafting operations on the tree; these operations are collectively called *subtree replacement* operations. At all times during editing, an *editing cursor* points to an interior node of the semantic tree. A subtree replacement operation can be performed at the position in the tree indicated by the editing cursor. The cursor can be moved from a node to its parent, or from a node to its child, by cursor movement operations.

After a subtree replacement, some attributes may become inconsistent. An attribute is *inconsistent* if its value is not equal to its semantic function applied to the current values of its arguments. An incremental attribute evaluator reevaluates the inconsistent attributes, thus reestablishing consistency among the attributes in the tree.

Continuing with the example from the previous subsection, suppose that a programmer, Joe, was editing a program containing the two declarations of *a* and *b*. If he were to add the following line to his program:

```
    a: character;
```

then, after attribute reevaluation, the value of the *error* attribute associated with this declaration would be "<-- Variable already declared". Such attributes can be displayed by the editor as part of the program text to notify the programmer of inconsistencies in the program. So Joe would see the following on his display after making the change:

```
    a: character; <-- Variable already declared
```

This illustrates how change analysis and change propagation are accomplished by means of attribute evaluation. It is desirable that the evaluation strategy be incremental, that is, it does not perform an exhaustive evaluation of all the attributes in the semantic tree, but only reevaluates those that are affected by the change.

The problem of incremental attribute evaluation for single subtree replacements can be stated as follows. Starting from a consistently attributed tree T, a subtree S of T is replaced by another tree, S', which is also consistently attributed. The root node of the two subtrees, S and S', must be labeled with the same nonterminal grammar symbol. Let T' be the tree T with S replaced by S'. The problem is to evaluate the minimum number of attributes in T' so that attribute consistency is reestablished. An optimal solution to this problem for the general class of noncircular AGs was devised by Reps, Teitelbaum and Demers [Reps 83].

Before describing this optimal algorithm, we define some terminology and state the assumptions used. We assume that the AG is in Bochmann normal form.[6] A replacement of subtree S with root node r by a subtree S' with root node r' consists of pruning the subtree S from the tree, assigning the inherited attributes of r' to the inherited attributes of r, and grafting the subtree S' onto r.[7] Thus, initially, the only inconsistent attributes are those associated with the root of the replaced subtree.

[6]This assumption is made only to simplify the exposition, and the algorithm to be described can be easily extended to deal with grammars that are not in normal form.

[7]Recall that nodes r and r' must be labeled with the same nonterminal symbol.

After a subtree replacement at node r, the algorithm starts by evaluating the attributes associated with r. The problem is to determine in what order the attributes of r should be evaluated. It is not sufficient to consider just the direct dependencies among the attributes of r in the two production instances where the node r appears. The reason is that although there may not be any direct dependencies among these attributes, they may be linked through a chain of dependencies arbitrarily far down the subtree rooted at r, or in the tree above the node r. (See, for example, the dependencies between the attributes *SymTabIn* and *SymTabOut* of *Decls* in figure 2-2 (c).) For this reason, the scheduling algorithm for determining which attribute should be evaluated next must take into account transitive dependencies.

For the general class of noncircular AGs, the transitive dependencies among the attributes of a nonterminal symbol may be different for different occurrences of the symbol in a semantic tree. The *upper tree context* of a nonterminal instance r in a semantic tree is the resulting tree after the subtree rooted at r is pruned. There can be several different subtrees derived from r, and several upper tree contexts. Therefore, it is in general not possible to determine the transitive dependencies among the attributes of a nonterminal symbol from a static analysis of the AG.

For the single subtree replacement case, Reps *et al.* use characteristic graphs to keep track of transitive dependencies among the attributes of a nonterminal occurrence in the tree. A *characteristic graph* is a directed graph, $G = (V, E)$, where V consists of the attribute instances associated with a nonterminal occurrence in the semantic tree, and an edge (v, w) is in E, where v and w are in V, and there is a path from v to w in the dependency graph of the semantic tree that does not go through any other attributes in V. A *subordinate* characteristic graph of a node r, denoted by $r.C$, only considers dependencies in the subtree rooted at r. A *superior* characteristic graph of a node r, denoted by $r.\overline{C}$, only considers dependencies in the upper tree context of r.

We first define the graph operations, union ($\cup$), deletion ($-$), and projection ($/$), that are used in the incremental attribute evaluation algorithm.

- Given directed graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, which may or may not be disjoint, the *union* of G_1 and G_2 is defined as:

$G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$

- The *deletion* of G_2 from G_1 is defined as:

  $G_1 - G_2 = (V_1, E_1 - E_2)$

- Given a directed graph $G = (V, E)$ and a set of vertices $V' \subseteq V$, the *projection* of G onto V' is defined as:

  $G / V' = \{(v, w) \mid v, w \in V'$ and there exists a path from v to w in G that does not contain any elements of $V'\}$.

Figure 2-3 shows the incremental algorithm for reevaluating a semantic tree T after a subtree replacement at r has occurred [Reps 84b]. It makes use of two central data structures: (1) a model M, which is a graph containing attributes that need reevaluation and direct and transitive dependency edges among them, and (2) a worklist S, which contains those attributes in the model that are ready to be evaluated (*i.e.*, their arguments have already been evaluated, or do not need to be reevaluated). M initially contains the attributes of the root of the replaced subtree, r. The (direct and transitive) dependencies among these attributes are obtained from the characteristic graphs associated with r. Those attributes in M that have no incoming edges are placed in the worklist S.

Attributes are removed from S and evaluated until S is empty. When an attribute is evaluated and its value changes, other attributes that depend on it may need to be brought into the model. This is performed by the EXPAND procedure, shown in figure 2-4 and explained below. The attribute that was just evaluated, as well as all its outgoing edges, are then removed from the model. This may result in some additional attributes becoming ready for evaluation, which are inserted into the worklist S.

The model is expanded by procedure EXPAND when an attribute b is reevaluated and changes value, and b has a successor c that is not in the model. EXPAND adds to the model all the attributes of a neighboring production of the attribute b, as well as all dependency edges among them. Let p denote the production $X_0 \rightarrow X_1 \cdots X_n$, and $D(p)$ the direct dependencies of the production p. If b is associated with X_0 (so c is associated with some child X_i, $1 \leq i \leq n$, of X_0), then the model is expanded downwards by production p to include all the attributes of p. If b is associated with X_i, $1 \leq i \leq n$

```
procedure PROPAGATE(T: semantic tree; r: nonterminal node of T at root of replaced subtree);
declare
    M: a directed graph;
    S, NeedToBeEvaluated: sets of attribute instances;
    b, c: attribute instances;
    changed: Boolean;
    OldValue, NewValue: attribute values;
begin
    M := r.C ∪ r.C̄ ;
    S := the set of vertices of M with in-degree 0 in M;
    NeedToBeEvaluated := the set of vertices of M;
    while S ≠ ∅ do
        Select and remove a vertex b from S;
        changed := false
        if b ∈ NeedToBeEvaluated then
            Remove b from NeedToBeEvaluated;
            OldValue := value of b;
            evaluate b;
            NewValue := value of b;
            if OldValue ≠ Newvalue then
                changed := true;
                if M does not contain all the successors of b in D(T) then
                    EXPAND(M, b, S)
                fi
            fi
        fi
        while there exists c, a successor of b in M do
            Remove edge (b, c) from M;
            if in-degree of c in M is 0 then Insert c into S fi
            if changed = true then Insert c into NeedToBeEvaluated fi
        od
    od
end
```

Figure 2-3: *Reps' Incremental Attribute Evaluation Algorithm*

(so c is associated with the parent node X_0 or a sibling node X_j, $1 \leq j \leq n, j \neq i$), then the model is expanded upwards by production p. Direct and transitive dependency edges among the attributes of production p are added to the model, using the two functions *ExpandSubordinate* and *ExpandSuperior* defined below.

$$ExpandSubordinate(X_0) \equiv D(p) \cup X_1.C \cup \cdots \cup X_n.C$$

```
procedure EXPAND(M: a directed graph; b: an attribute instance; S: a set of attribute instances);
declare
    c: an attribute instance;
begin
    if there exists c, a successor of b in D(T) that is not in M
            and TreeNode(c) is a child of TreeNode(b) then
        M := (M − TreeNode(b).C) ∪ ExpandedSubordinate(TreeNode(b));
        Insert into S all vertices of ExpandedSubordinate(TreeNode(b)
                        whose in-degree in M is 0
    fi
    if there exists c, a successor of b in D(T) that is not in M
            and TreeNode(c) is the parent or a sibling of TreeNode(b) then
        M := (M − TreeNode(b).C̄) ∪ ExpandedSuperior(TreeNode(b));
        Insert into S all vertices of ExpandedSuperior(TreeNode(b)
                        whose in-degree in M is 0
    fi
end
```

Figure 2-4: *Expanding a Model*

$$ExpandSuperior(X_i) \equiv D(p) \cup X_0.\overline{C} \cup X_1.C \cup \cdots \cup X_{i-1}.C \cup X_{i+1}.C \cup \cdots \cup X_n.C$$

Since the model is expanded by an entire production instance, an attribute may be added to the model, and eventually evaluated, even if none of its arguments changes. To avoid this, a third data structure, *NeedToBeEvaluated*, is used in the incremental evaluation algorithm of figure 2-3 to keep track of the set of attributes in the model that depend on one or more changed attributes. Only attributes in *NeedToBeEvaluated* are evaluated.

Reps' incremental evaluation algorithm does not require both superior and subordinate characteristic graphs to be maintained at each node in the semantic tree. After a subtree replacement at a node r, PROPAGATE never needs subordinate characteristic graphs for nodes on the path from r to the root of the tree, and it never needs superior graphs for any node that is not on the path from r to the root. Reps defines the "prepared for propagation" invariant on the semantic tree representing the program being edited as follows:

- The nonterminal node where the editing cursor is placed is labeled with both its superior and subordinate characteristic graphs.

- Each node on the path from the cursor to the root of the tree is labeled with its superior characteristic graph.
- Each node that is not on the path from the cursor to the root of the tree is labeled with its subordinate characteristic graph.

The prepared for propagation invariant is reestablished after subtree replacement operations and when the cursor is moved. The functions *ComputeSubordinate* and *ComputeSuperior* compute the subordinate and superior characteristic graphs of a semantic tree node, respectively, and are defined as follows for nodes in the production $p: X_0 \rightarrow X_1 \cdots X_n$.

$$\mathit{ComputeSubordinate}(X_0) \equiv \mathit{ExpandSubordinate}(X_0) \,/\, A(X_0)$$

$$\mathit{ComputeSuperior}(X_i) \equiv \mathit{ExpandSuperior}(X_i) \,/\, A(X_i)$$

For a given grammar, the functions *ComputeSubordinate* and *ComputeSuperior* each have unit cost.

Reps' incremental evaluation algorithm for a single subtree replacement is asymptotically optimal in time. This means that both the number of semantic function applications, as well as the cost of maintaining the necessary characteristic graphs, are proportional to the size of the set *AFFECTED*, where *AFFECTED* is the set of attributes whose values differ in T and T'.

Chapter 3

Incremental Attribute Evaluation for Multiple Asynchronous Subtree Replacements

In this chapter, we present new incremental attribute evaluation algorithms that handle multiple asynchronous subtree replacements for the general class of noncircular (well-defined) attribute grammars. Multiple asynchronous subtree replacements arise naturally in multi-user semantics-based editors as multiple programmers make changes simultaneously in different parts of the program. Incremental evaluation of asynchronous subtree replacements for restricted classes of attribute grammars is addressed in the following chapter.

When development or maintenance of a software system involves teams of programmers, the semantic tree representing the system being developed or maintained is divided into parts, called *segments*. The entire semantic tree is not usually available in main memory since segments may be edited on different workstations, or they may be dormant (that is, no one is currently working on them). In both this chapter and the next, the problem of incremental attribute evaluation for asynchronous subtree replacements is addressed separately from the issues of segmentation of the semantic tree, which are the topics of chapters 5, 6 and 7. The incremental evaluation algorithms to be described are applicable to either a nonsegmented semantic tree,[8] or to a segment of a segmented semantic tree. Multiple evaluation processes may be initiated asynchronously within a segment by (1) internal subtree replacements within the segment, and/or (2) updates to ''interface attributes'' of the segment caused by propagation from external subtree replacements within other segments of the tree.

[8]Although this scenario may not be practical for our primary application of multi-user programming environments, it may occur in other applications.

3.1. Problem Formulation

We state the problem of incremental attribute evaluation for k asynchronous subtree replacements for the case when k is two. Let T be a consistently attributed semantic tree of some attribute grammar G, T' the resulting tree after subtree S in T is replaced by S', and T'' the resulting tree after subtree R in T' is replaced by R', where the subtrees S' and R' are also consistently attributed. The two modifications at S and R are *asynchronous*, that is, the second one may occur while the evaluation of the first one is still in progress. The problem is to design an incremental evaluation algorithm that reestablishes attribute consistency in T'' in an optimal way.

The ''goodness'' of an incremental attribute evaluation algorithm is measured by two costs: (1) the number of attributes reevaluated, and (2) the bookkeeping costs incurred in scheduling attributes for reevaluation. An incremental evaluator for asynchronous subtree replacements is optimal if it meets the following requirements:

1. For any single modification, the algorithm evaluates only those attribute instances affected by the modification.
2. For any $k > 1$ modifications affecting the same attribute a, where the k evaluation processes are still in progress and none have yet evaluated a, the algorithm evaluates a at most once.
3. The bookkeeping costs of the evaluation algorithm are proportional to the number of attributes evaluated.

This is an ideal definition of optimality, and it remains an open question whether an algorithm that achieves all these requirements can be designed. There seems to be a tradeoff between minimizing (1) the number of attributes reevaluated, and (2) the bookkeeping costs incurred in scheduling attributes for reevaluation. Algorithms that compromise between these two costs in different ways are presented in this chapter and the next. That is, a particular evaluation algorithm may evaluate the minimum number of attributes but only by being sub-optimal in its bookkeeping costs. Such an algorithm would be useful for an application where attribute evaluation is expensive compared to the bookkeeping costs, such as the proof checker described in [Reps 84c].

The second requirement is the more important one for the purposes of this section of the thesis, so we shall state it more formally for the case when $k = 2$. Suppose that subtree S is replaced at time t_1, and subtree R at time t_2, where $t_1 \leq t_2$. Let $AFFECTED_S$ be the set of attributes that are affected (and therefore must be reevaluated) because of the subtree replacement at S, and similarly, $AFFECTED_R$ the set of attributes affected by the subtree replacement at R. Furthermore, suppose that the evaluations from the two modifications overlap, that is,

$$AFFECTED_S \cap AFFECTED_R \neq \emptyset$$

If the evaluation due to the subtree replacement at S is still in progress at the time of the second modification, t_2, then $AFFECTED_S$ can be divided into two subsets: (1) *EVAL*, containing those affected attributes that have already been evaluated at the time of the second replacement, and (2) *UNEVAL*, containing the attributes still needing evaluation.

$$AFFECTED_S = EVAL_{S,t_2} \cup UNEVAL_{S,t_2}$$

Note that all these sets are not known *a priori* but are determined as the evaluation is proceeding. The second optimality requirement states that every attribute a, such that

$$a \in UNEVAL_{S,t_2} \cap AFFECTED_R,$$

is evaluated at most once. The attribute a does not need to be evaluated at all if the change to the value of the attribute at the root of the subtree S that affected a was undone by the subtree replacement at R.

3.2. A Naive Algorithm for Multiple Updates

A naive approach to handling k asynchronous subtree replacements is to perform k sequential applications of the classical algorithm for single subtree replacements (described in chapter 2, section 2.2), where new changes and thus new evaluation processes are blocked until the previous one has completed.

This solution has two shortcomings. First, the classical algorithm assumes that there is a single editing cursor (*i.e.*, semantic tree node where a subtree replacement can occur), and is prepared for propagation at only one position in the tree — the position indicated by the (single) editing cursor. If a subtree replacement is performed at a semantic tree node different from the cursor, the characteristic graphs required by the evaluation process are not available. Therefore, to correctly handle k asynchronous edits at k cursors (for k users), the k sequential applications of the single-edit algorithm must be interleaved with commands to move a special internal cursor from the root of one subtree replacement to the next, making it appear as if there was only a single user cursor.

Second, if the sets of attributes affected by the k subtree replacements, $AFFECTED_i$, $1 \le i \le k$, are not disjoint, the naive solution would always evaluate attributes in the intersection more than once, even though information might have been available to prevent unnecessary evaluations.

3.3. Collision-Merging Algorithm for Multiple Updates

We present a better approach for handling k asynchronous subtree replacements. The k evaluation processes proceed independently while they cover disjoint parts of the semantic tree, where each independent process uses a variation of the classical algorithm for single subtree replacement. A model is maintained for each independent evaluation process in progress to schedule inconsistent attribute instances for reevaluation.[9] As the model changes, either when it is first initialized or when it is expanded, it may cover a semantic tree node that is already included in another model arising from a different subtree replacement that is being evaluated concurrently; this is called a *collision*. When this happens, the colliding models are merged into one, thereby combining the corresponding evaluation processes that had been proceeding independently prior to the collision, and evaluation continues with the merged model. We call this algorithm for incrementally evaluating a semantic tree for k asynchronous subtree replacements the *Collision-Merging Algorithm*.

Figures 3-1 and 3-2 show the modified classical incremental evaluation algorithm executed by each independent evaluation process. The parts of the algorithm that detect collisions and merge the colliding models are explained in later sections, as indicated in the figures.

When a subtree replacement occurs, evaluation processes in progress are suspended

[9]This is the same data structure used in the single subtree replacement algorithm presented in section 2.2.

```
atomic procedure startup ( T: semantic tree; r: node of T at root of replaced subtree );
declare
    M: a directed graph;
    S, NeedToBeEvaluated: sets of attribute instances;
begin
    if collision during model initialization is detected then
        Merge initial model with existing model /* defined in section 3.7 */
    else
        $M := r.C \cup r.\overline{C}$ ;
        S := the set of vertices of M with in-degree 0 in M;
        NeedToBeEvaluated := the set of vertices of M;
        fork asynch-propagate(M, S, NeedToBeEvaluated)
    fi
end
```

Figure 3-1: *Startup Algorithm*

until the pruning and grafting operations on the tree have been performed, and an evaluation process for the new subtree replacement initiated. A new evaluation process is initiated by the procedure *startup*, shown in figure 3-1.

An evaluation process can only be suspended when it is not executing a critical section. The *startup* procedure, which initializes the model and other data structures used by the evaluation process, is a critical section, and is defined as an atomic operation. The second critical region consists of each iteration of the while loop body of procedure *asynch-propagate* shown in figure 3-2; this critical section removes an attribute *b* from the worklist *S*, evaluates *b*, and if necessary expands the model.

The novelty of the Collision-Merging algorithm is not the approach itself, which was originally reported by Kaplan and Kaiser [Kaplan 86] and improved and expanded in [Kaiser 90]. Rather, our contribution is the complete collection of supporting algorithms needed to make the approach work. In particular, we describe solutions to the following problems previously left open:

- How are superior and subordinate characteristic graphs maintained in the semantic tree when there are multiple editing cursors at which subtree replacements can be initiated?
- What constitutes a collision between models of independent evaluation processes, and when do collisions occur?

```
procedure asynch-propagate ( M: a directed graph; S, NeedToBeEvaluated: sets
                                        of attribute instances );
declare
    b, c: attribute instances;
    changed: Boolean;
    OldValue, NewValue: attribute values;
begin
    while $S \neq \emptyset$ do
        atomic begin
            Select and remove a vertex b from S;
            changed := false
            if $b \in$ NeedToBeEvaluated then
                Remove b from NeedToBeEvaluated;
                OldValue := value of b;
                evaluate b;
                NewValue := value of b;
                if OldValue $\neq$ Newvalue then
                    changed := true;
                    if M does not contain all the successors of b in D(T) then
                        asynch-expand(M, b, S) /* defined in sections 3.6 and 3.7 */
                    fi
                else
                    antifreeze(b, M, S) /* defined in section 3.7 */
                fi
            fi
            while there exists c, a successor of b in M do
                Remove edge (b, c) from M;
                if in-degree of c in M is 0 then Insert c into S fi
                if changed = true then Insert c into NeedToBeEvaluated fi
            od
        atomic end
    od
    clear-model-field(apex of M) /* defined in section 3.6 */
end
```

Figure 3-2: *Propagate Algorithm for Asynchronous Subtree Replacements*

- Following a collision, how are the models merged to ensure that the tree is consistently attributed when the evaluation process using the merged model terminates?

As mentioned earlier, there is a trade-off between minimizing the number of attributes reevaluated and bookkeeping costs for the multiple asynchronous subtree replacement problem. This results in a spectrum of algorithms that differ in how they compromise between these costs. In the naive approach described earlier, bookkeeping is simplified at the expense of reevaluating attributes more than once; this approach lies at one end of the spectrum.

A different approach at the other end of the spectrum consists of incrementally maintaining a complete transitive closure of the entire dependency graph; the number of attributes reevaluated is minimized at the expense of much higher bookkeeping costs. The Collision-Merging algorithm attempts to find a balance between the two costs: additional bookkeeping costs are only incurred at collision, and on average, collisions prevent unnecessary attribute reevaluations. Furthermore, the Collision-Merging algorithm is easily extended to actually minimize attribute reevaluations, with the concomitant higher bookkeeping costs, if necessary.

The rest of this chapter is organized as follows. Section 3.4 introduces terminology. The additional complexity introduced by multiple editing cursors is explored in section 3.5. In section 3.6, the scenarios where collisions occur are described, and algorithms for merging models involved in a collision are presented in section 3.7. The cost of the Collision-Merging algorithm is analyzed in section 3.8. Related work is compared in section 3.9.

3.4. Terminology

The model — the graph for scheduling attribute evaluations introduced in section 2.2 — represents direct and transitive dependencies of a connected region of the semantic tree. Its purpose is to prevent an attribute from being considered for evaluation before its arguments receive their final values. This ensures that an attribute is not evaluated more than once due to a single subtree replacement. When a subtree replacement occurs, the only inconsistent attributes initially are those associated with the root *r* of the replaced subtree.[10] Thus, the initial model consists of the attributes of *r* and the dependencies among them (both direct and transitive).

[10]We assume that the attribute grammar is in Bochmann normal form.

When an attribute b is evaluated and its value changes, the modeled region of the tree may need to be expanded. Expansion is required if there is an attribute c that depends on b, and c is not in the model. In this case, the modeled region of the tree is expanded to include the production p, where c is defined, by adding to the model the attributes associated with nonterminal instances of p, as well as direct and transitive dependencies among them. Since semantic equations have locality of reference, p is adjacent to the previously modeled region of the semantic tree.

The modeled region of the semantic tree, which we call the *backbone* of the model, is a rooted subgraph of the semantic tree; it includes the root node of the replaced subtree and all production instances by which the model was expanded. The attribute instances in the model are associated with the tree nodes in the backbone, but not all attribute instances in the backbone are in the model — some of them may have been deleted, either because they were evaluated or because it was not necessary to evaluate them because their arguments did not change. We call the root node of the backbone the *apex*, and the leaves of the backbone the *frontier*. The root and leaf nodes of the backbone do not necessarily correspond to the root and leaf nodes of the semantic tree.

A model contains two kinds of edges: direct dependencies, and transitive dependencies. There is a direct dependency edge (a,b) between two attributes a and b if a is an argument to the semantic function defining b. The model only represents transitive dependency edges between attributes of the same nonterminal instance. There is a transitive edge (a,b) between two attributes a and b of nonterminal X if there is a path of direct dependency edges between a and b. If the path only goes through attributes associated with nodes in the subtree rooted at X, then this is a *subordinate* transitive edge. If the path only goes through attributes associated with nodes in the tree that remains after the subtree rooted at X is pruned, then the edge is called a *superior* transitive edge. There are superior transitive dependency edges among the attributes associated with the apex of the model, subordinate transitive dependencies among the attributes of frontier nodes of the model, and direct dependency edges everywhere else.

3.5. Multiple Cursors and Characteristic Graphs

In this section we describe how to correctly maintain the characteristic graphs associated with semantic tree nodes for multiple editing cursors. Characteristic graphs are employed to efficiently initialize and expand the model used by each independent evaluation process of the Collision-Merging algorithm. Characteristic graphs, defined in section 2.2, contain the transitive dependencies among the attribute instances of a semantic tree node.

When there are multiple editing cursors, the semantic tree is "prepared for propagation" at any one of the editing cursors if the following invariant holds. Let m be the number of editing cursors, $r_1, \ldots, r_m$ denote the m cursors, and *LCA* their lowest common ancestor. Then, the invariant, which is illustrated in figure 3-3, states that:

- Each node on the path from the *LCA* (inclusive) to the root of the tree is labeled with its superior characteristic graph.
- Each node on the path from an editing cursor r_i (inclusive) to the *LCA* (not inclusive) is labeled with both its superior and subordinate characteristic graphs, where $1 \leq i \leq m$.
- Each node that is not on a path from an editing cursor to the root of the tree is labeled with its subordinate characteristic graph.

For a segmented semantic tree, the prepared for propagation invariant is defined with respect to each segment, and not for the entire semantic tree. Within a segment, there is a (real) editing cursor for the single user making changes to that segment, and (virtual) editing cursors for each "interface" node of the segment (that is, a node on the boundary with another segment). We treat an interface node of a segment as an editing cursor because an update in some other segment of the tree is propagated to the segment via an interface node, and is handled as a subtree replacement at that interface node. (The details of the attribute evaluation algorithm for a segmented semantic tree are explained in chapter 6, section 6.1.)

In the rest of this section, we examine the two events that affect the prepared for propagation invariant for multiple editing cursors: cursor movement operations, and subtree replacements. When an editing cursor is moved to a neighboring node in the

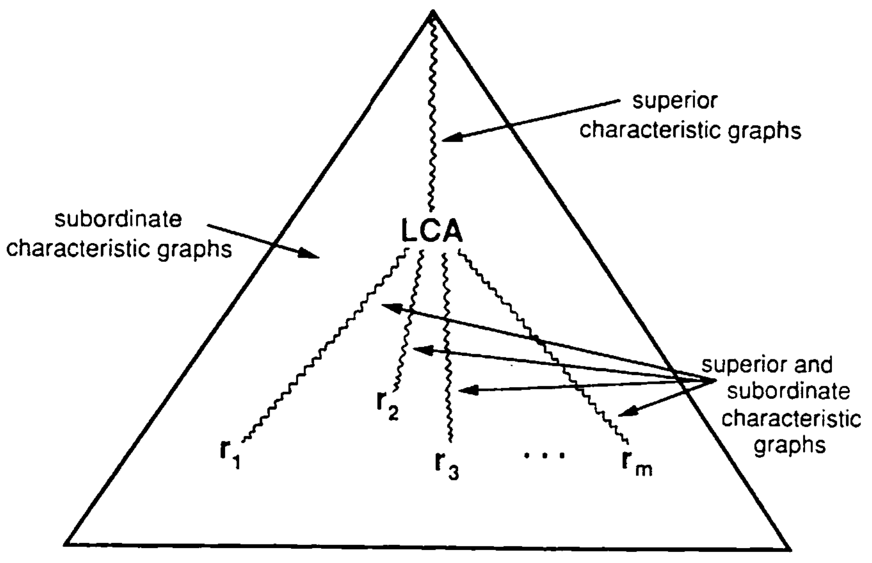

superior characteristic graphs
subordinate characteristic graphs
LCA
superior and subordinate characteristic graphs
r_1
r_2
r_3
. . .
r_m

Figure 3-3: *Characteristic Graphs Required for Multiple Editing Cursors*

semantic tree, characteristic graphs may have to be computed or deleted to maintain the prepared for propagation invariant; this issue is the topic of subsection 3.5.1. A subtree replacement at one of the editing cursors may invalidate characteristic graphs in other parts of the semantic tree; algorithms to recompute the necessary characteristic graphs after a subtree replacement are described in subsection 3.5.2.

3.5.1. Effect of Cursor Movement on Characteristic Graphs

Every cursor movement can be broken up into a sequence of two kinds of cursor movement operations: (1) *DescendToChild*(r_i,j) moves cursor r_i to the j th child of r, and (2) *AscendToParent*(r_i) moves cursor r_i to its parent. We assume that the cursor movement operations, *DescendToChild* and *AscendToParent*, are atomic.

The actions that must be performed to reestablish the prepared for propagation invariant when an editing cursor r_i is moved depends on whether (1) r_i is the *LCA*, (2) r_i is not the *LCA* but is the ancestor of another editing cursor r_j, $1 \leq j \leq m, j \neq i$, or (3) r_i is not the ancestor of any other editing cursor r_j. These three cases are mutually exclusive when there are multiple editing cursors (*i.e.*, $m > 1$).

To maintain the prepared for propagation invariant, *DescendToChild*(r_i, j) performs the following actions:

- Case $(r_i = LCA)$:
  - a. Compute new *LCA*.
  - b. If new *LCA* is not equal to r_i (*i.e.*, new *LCA* is equal to j th child of r_i), delete subordinate characteristic graph of r_i.
- Case $(\exists j,\ 1 \le j \le m, j \ne i, r_i$ is ancestor of $r_j)$:
  - a. Do nothing (superior characteristic graph of j th child of r_i is already available).
- Case $(\forall j,\ 1 \le j \le m, j \ne i, r_i$ is not ancestor of $r_j)$:
  - a. Compute superior characteristic graph of j th child of r_i using function *ComputeSuperior* defined in chapter 2, section 2.2.

AscendToParent(r_i) reestablishes the prepared for propagation invariant as follows:

- Case $(r_i = LCA)$:
  - a. Set new *LCA* to be parent of r_i.
  - b. Compute subordinate characteristic graph of parent of r_i using function *ComputeSubordinate* defined in chapter 2, section 2.2.
- Case $(\exists j,\ 1 \le j \le m, j \ne i, r_i$ is ancestor of $r_j)$:
  - a. Do nothing (superior characteristic graph of r_i still needed).
- Case $(\forall j,\ 1 \le j \le m, j \ne i, r_i$ is not ancestor of $r_j)$:
  - a. Delete superior characteristic graph of r_i.

The actions performed by the cursor movement operations to reestablish the prepared for propagation invariant depend on whether an editing cursor is an ancestor of another cursor, and may entail the recalculation of *LCA*. We now describe how this can be done efficiently so that for a given grammar, each cursor movement operation has unit cost.

We maintain an ``edge-color'' invariant on the semantic tree, which is defined as follows:

- Each semantic tree edge on the path from an editing cursor to the *LCA* is colored blue.
- Each edge that is not on a path from an editing cursor to the *LCA* is colored white.

Given the edge-color invariant, we can determine whether a semantic tree node n is a (proper) ancestor of an editing cursor by checking whether there is a blue edge from n to any child of n. Furthermore, we can easily recompute the *LCA* when an editing cursor r_i that is positioned at the *LCA* is moved to a child c — if there is a blue edge from r_i to any sibling of c, then *LCA* stays the same, otherwise *LCA* is set to c.

The edge-color invariant must be reestablished after each cursor movement operation. When an editing cursor r_i is moved to a child c, there are two situations when the color of the edge (r_i, c) must be changed. (1) If r_i is not an ancestor of any other editing cursor, then the (white) edge (r_i, c) is colored blue. (2) If r_i is positioned at the *LCA*, and moving r_i to c changes *LCA* to c, then the (blue) edge (r_i, c) is colored white. The part of *DescendToChild*(r_i, j) that maintains the edge-color invariant is defined as:

```
let $c = j$ th child of $r_i$ in
    if ($r_i = LCA$) and (there is no blue edge from $r_i$ to a child $k$ of $r_i$, where $k \neq j$) then
        $LCA := c$;
        Color edge ($r_i$, $c$) white
    else
        if there is no blue edge from $r_i$ to any child of $r_i$ then
            Color edge ($r_i$, $c$) blue
        fi
    fi
ni
```

Reestablishing the edge-color invariant when an editing cursor r_i is moved to its parent p is similar. Again, there are two situations when the color of the edge (p, r_i) must be changed. (1) If r_i is not an ancestor of any other editing cursor, then the (blue) edge (p, r_i) is colored white. (2) If r_i is positioned at the *LCA*, then the (white) edge (p, r_i) is colored blue. The part of *AscendToParent*(r_i) that maintains the edge-color invariant is defined as:

```
let $p$ = parent of $r_i$ in
    if ($r_i = LCA$) then
        $LCA := p$;
        Color edge ($p$, $r_i$) blue
    else
        if there is no blue edge from $r_i$ to any child of $r_i$ then
            Color edge ($p$, $r_i$) white
        fi
    fi
ni
```

Putting this all together, we obtain an efficient (unit cost) algorithm for reestablishing the prepared for propagation invariant when a cursor is moved. The resulting algorithms for the two kinds of cursor movement operations are shown in figure 3-4.

3.5.2. Updating Characteristic Graphs after Subtree Replacement

A subtree replacement at one of the editing cursors may invalidate characteristic graphs that may be needed for evaluations initiated by subsequent, or previously started but still in progress, evaluation processes. Consider the tree illustrated in figure 3-5 (a), with two editing cursors at the nodes labeled X and Y. The superior and subordinate characteristic graphs of all nodes in the tree contain two vertices, attributes a and b, and no edges. If the subtree whose root is Y is replaced by a new subtree in which there is a direct dependency between the attributes a and b of Y, as shown in figure 3-5 (b), the superior characteristic graphs of the nodes on the path from X to the root should now contain an edge from b to a (and from a to b for the subordinate graphs on the path from Y to the root).

Consider the possibility that the characteristic graphs are not updated after subtree replacements. If a subtree replacement at node X updated the value of X's attribute b, this change might not be reflected consistently in all attributes depending on b because the model does not have any transitive dependency edges from b to a for nodes on the path from X to the root (and from a to b for nodes on the path from Y to the root). The consequence might be that the a attributes may be evaluated before the b attributes for the left path and vice versa for the right path. Because the model is supposed to prevent an attribute from being assigned a value that is not its final value, once an attribute has

```
DescendToChild($r_i$, $j$):
    let $c$ = $j^{th}$ child of $r_i$ in
        if ($r_i$ = LCA) and (there is no blue edge from $r_i$ to a child $k$ of $r_i$, where $k \neq j$) then
            LCA := $c$;
            Color edge ($r_i$, $c$) white;
            $r.C$ := null
        else
            if there is no blue edge from $r_i$ to any child of $r_i$ then
                $c.\overline{C}$ := ComputeSuperior($c$);
                Color edge ($r_i$, $c$) blue
            fi
        fi
    ni

AscendToParent($r_i$):
    let $p$ = parent of $r_i$ in
        if ($r_i$ = LCA) then
            LCA := $p$;
            Color edge ($p$, $r_i$) blue;
            $p.C$ := ComputeSubordinate($p$)
        else
            if there is no blue edge from $r_i$ to any child of $r_i$ then
                Color edge ($p$, $r_i$) white;
                $r_i.\overline{C}$ := null
            fi
        fi
    ni
```

Figure 3-4: *Reestablishing Prepared for Propagation Invariant for Multiple Cursors*

been removed from the model and evaluated, it is never considered for evaluation again. Thus, some of the attributes (in particular, the a attributes on the left path, and the b attributes on the right path), may be incorrect when evaluation terminates.

Let $path_1, \ldots, path_m$ denote the m paths from the editing cursors $r_1, \ldots, r_m$ to *LCA*. Note that these paths are not necessarily node disjoint, and it is possible for a path to have zero length in the case when the *LCA* is itself pointed to by one of the editing

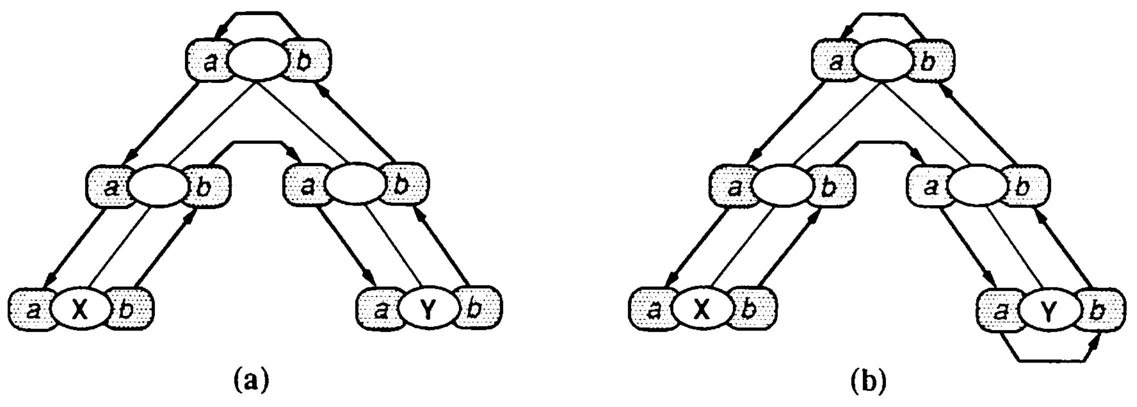


Figure 3-5: *Changing Transitive Dependencies*

cursors. In general, a subtree replacement at a cursor r_i, $1 \leq i \leq m$, may invalidate subordinate characteristic graphs associated with nodes on $path_i$, and superior characteristic graphs associated with those nodes on paths $path_j$ that are disjoint from the nodes on $path_i$, where $1 \leq j \leq m$ and $j \neq i$.

An algorithm that updates the necessary characteristic graphs after a subtree replacement is shown in figure 3-6. This algorithm is atomic, and is invoked when a subtree replacement at a node *r* occurs, before the evaluation process is initiated. Note that this procedure only needs to be called if the subtree replacement changes the subordinate characteristic graph of *r*.

To explain procedure *cgraph_update*, we recall the definition of the functions *ComputeSubordinate* and *ComputeSuperior* from chapter 2, section 2.2. The subordinate characteristic graph of a node *n* is used in the computation of the subordinate characteristic graph of the parent of *n*, and in the computation of the superior characteristic graphs of siblings of *n*. Therefore, if the subordinate characteristic graph of *n* changes, these characteristic graphs must be recomputed. The superior characteristic graph of a node *n* is used in the computation of the superior characteristic graphs of children of *n*, which must be recomputed if the superior characteristic graph of *n* changes.

Procedure *cgraph_update* is driven by a new value for the subordinate characteristic

```
        atomic procedure cgraph_update ( r: semantic tree node at root of subtree replacement );
        declare
            node, s: semantic tree nodes;
            new_C: characteristic graph;
            changed: boolean;
        begin
[1]         if ($r \neq LCA$) then
[2]             node := parent(r);
[3]             changed := true;
[4]             while ($node \neq LCA$) and (changed) do
[5]                 new_C := ComputeSubordinate(node);
[6]                 if (node.C = new_C) then changed := false
[7]                 else
[8]                     node.C := new_C;
[9]                     for each sibling s of node do
[10]                        superior_update(s);
[11]                    od
[12]                    node := parent(node)
[13]                fi
[14]            od
[15]            if (changed) and (LCA is pointed to by an editing cursor) then
[16]                node.C := ComputeSubordinate(node)
                fi
            fi
        end /* of cgraph_update */
```

Figure 3-6: *Algorithm to Update Incorrect Characteristic Graphs*

graph of *node*, computed in line [5]. The superior characteristic graphs of siblings of *node* are updated by a call to the procedure *superior_update*, shown in figure 3-7 and explained below. The subordinate characteristic graph of the parent of *node* is recomputed in the next iteration of the while loop. This process is repeated for each node on the path from the root of the subtree replacement to the *LCA*, until the subordinate characteristic graph of a node does not change (line [6]), or the *LCA* is reached.

The *LCA* is treated as a special case (lines [15]-[16]). If the *LCA* is not pointed to by an editing cursor, then it is not labeled with a subordinate characteristic graph, and therefore nothing must be done. If the *LCA* is pointed to by an editing cursor, then its subordinate characteristic graph must be recomputed if the subordinate characteristic

```
      procedure superior_update ( node: semantic tree node );
      declare
         queue: FIFO list of semantic tree nodes;
         c, p: semantic tree nodes;
         new_$\overline{C}$: characteristic graph;
      begin
[1]      Initialize queue to contain node;
[2]      while (queue is not empty) do
[3]         Remove node p from front of queue;
[4]         new_$\overline{C}$ := ComputeSuperior(p);
[5]         if (p.$\overline{C}$ $\neq$ new_$\overline{C}$) then
[6]            p.$\overline{C}$ := new_$\overline{C}$;
[7]            for each child c of p do
[8]               if edge (p,c) is colored blue then Insert c into queue fi
               od
            fi
         od
      end /* of superior_update */
```

Figure 3-7: *Characteristic Graph Update (cont.)*

graph of a child of *LCA* changed. However, the siblings of *LCA* are not labeled with their superior characteristic graphs, and therefore no superior characteristic graphs need to be updated as a result of a change to the subordinate characteristic graph of *LCA*.

Procedure *superior_update*, shown in figure 3-7, recomputes the superior characteristic graph of a node *n*, and if necessary, the superior characteristic graphs of the descendents of *n*. It is necessary to recompute the superior characteristic graph of a descendent node if (1) the superior characteristic graph of *n* changes, and (2) the descendent node is labeled with its superior characteristic graph (*i.e.*, the descendent is on the path from an editing cursor to the *LCA*). The edge-color invariant is used to determine which descendent nodes are labeled with their superior characteristic graphs. The descendents of the node *n* are visited in a breadth-first manner. Traversal along a path terminates either when the superior characteristic graph of a node does not change (line [5]), or when a node that is not labeled with a superior characteristic graph is reached (line [8]).

The cost of updating characteristic graphs invalidated by a subtree replacement is at most proportional to the number of semantic tree nodes on the paths from the m editing cursors to the *LCA*. In the worst case, this cost can be $O(n)$. In practice, we expect that the semantic tree nodes whose characteristic graphs are changed because of a subtree replacement also have attribute instances that are affected by the subtree replacement, and therefore, the bookkeeping cost incurred to update characteristic graphs after a subtree replacement is proportional to the number of attributes affected by the subtree replacement.

3.6. Detecting Collisions

We now refine the definition of a collision given earlier in section 3.3. Two models are involved in a collision if the corresponding backbones of the models are not node disjoint (*i.e.*, the two models share a semantic tree node). There are two cases when models collide with each another in the course of attribute evaluation:

- During initialization of one of the models.
- During expansion of one of the models.

After a subtree replacement at a node r, the initial model created for the new evaluation process contains the attribute instances associated with node r and dependencies among them. The backbone of the initial model consists of the single node r. An *initialization collision* occurs if node r is in the backbone of another model used by a different evaluation process.

A model (and its associated backbone) can be expanded either downwards from a frontier node, or upwards from the apex. Consider the semantic tree illustrated in figure 3-8, where the backbones of two models M_1 and M_2 are shown. If M_1 is expanded downwards at node X, then the backbone for M_1 grows to include the production instance that applies at X. This production instance includes the node Y, which is the apex of the backbone of M_2. The result is a *downward collision* at node Y. In general, a downward expansion of the model at a node X_0, where the production that applies at X_0 is $X_0 \to X_1 \cdots X_n$, can have up to n downward collisions at the children of X_0.

A dual case arises if the model M_2 is expanded upwards from node Y, causing an

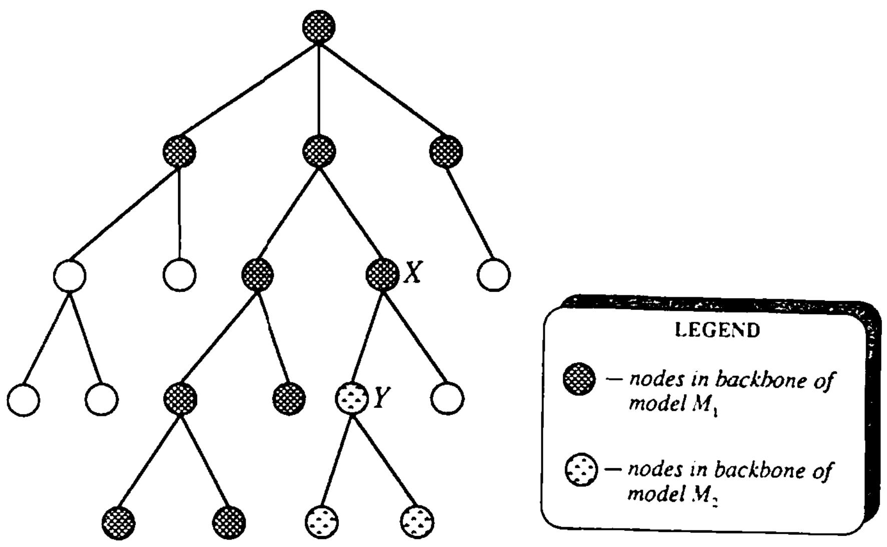

X
Y
LEGEND
— nodes in backbone of model M_1
— nodes in backbone of model M_2

Figure 3-8: *Scenario for Collisions during Model Expansion*

upward collision at node X. Generally, an upward expansion of the model at a node X_i, $1 \leq i \leq n$, by production $X_0 \rightarrow X_1 \cdots X_n$, can have an upward collision at the parent X_0 of the expansion node X_i, and up to $n - 1$ downward collisions at the siblings of X_i.

To detect collisions, each semantic tree node contains additional information indicating whether the node is at the apex, interior or frontier of the backbone of some model. This information is stored in the *in-model* field of each semantic tree node. The *in-model* field of a node n is equal to

- *apex/frontier* — if n is both the apex and a frontier node of the backbone of a model (*i.e.*, n is the backbone of an initial model).
- *apex* — if n is the apex of the backbone of a model.
- *interior* — if n is an interior node of the backbone of a model.
- *frontier* — if n is on the frontier of the backbone of a model.
- *not-in-model* — if n is not in the backbone of any model.

An initialization collision is detected if a subtree replacement at r occurs, and the *in-model* field of r is not equal to *not-in-model.* Initialization collisions must be detected before the old subtree is pruned from the tree. The reason is that some cleanup operations must be performed for the part of the existing model that covered the deleted subtree. (The details are explained later in section 3.7.)

Procedure *asynch-expand*, shown in figure 3-9, expands a model and also detects collisions during model expansion. If the model is expanding downwards at a node n, then each child of n is checked to determine whether the child node is an apex of another model, in which case a downward collision is detected. If the model is expanding upwards from n, then an upward collision is detected at the parent of n if the parent node is a frontier node of another model, and a downward collision is detected at a sibling of n if the sibling is an apex of another model. The elided parts of procedure *asynch-expand* merge the colliding models, and are elaborated in section 3.7.

When a model is expanded by a production p, the *in-model* field of semantic tree nodes of p must be updated, as shown in procedure *asynch-expand.*

When an evaluation process terminates, the *in-model* field of nodes in the backbone of the model must be cleared. This is accomplished by the atomic procedure *clear-model-field*, shown in figure 3-10. This procedure clears the *in-model* field of semantic tree nodes in the model's backbone during a depth-first traversal of the backbone, started from the backbone's apex.

3.7. Merging Colliding Models

After a collision is detected, the colliding models are ''merged'' into one. We discuss the problem of merging two colliding models for the case of a downward collision caused by a downward expansion of a model. Merging models subsequent to an upward collision caused by an upward expansion of a model is similar, and therefore omitted. Merging two models following an initialization collision is discussed in subsection 3.7.2.

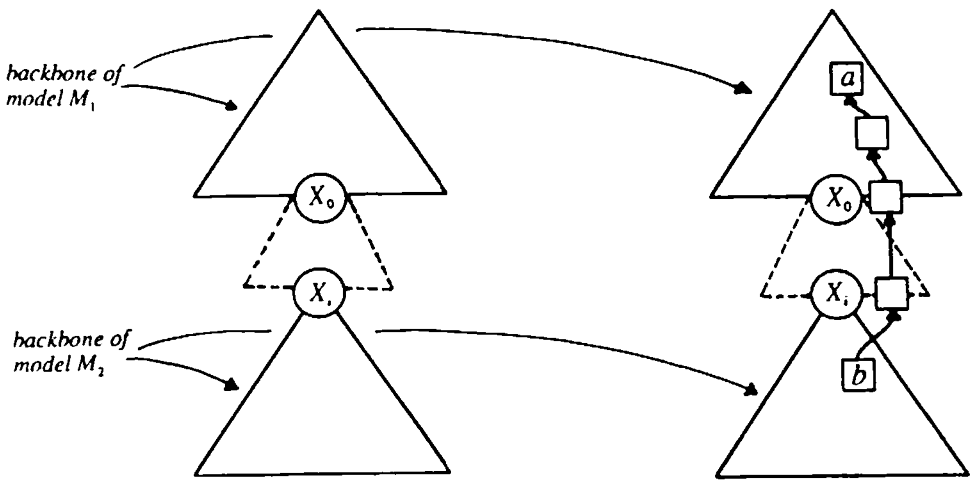
Figure 3-11 (a) illustrates a downward collision resulting from a downward expansion. Model M_1 expands downwards at node X_0 by production p: $X_0 \rightarrow X_1 \cdots X_n$, causing a collision with model M_2 at node X_i, where $1 \leq i \leq n$. If M_2 was not in the picture, the expansion would add to model M_1 the attributes of X_i, all direct edges between attributes of X_i and other attributes in p, and subordinate transitive edges among attributes of X_i (and similarly for each sibling of X_i). Let M_1^+ denote the expanded model M_1.

```
atomic procedure asynch-expand ( M: a directed graph; b: an attribute instance; S: set of
                                        attribute instances );
declare
  c: an attribute instance;
begin
  if there exists c, a successor of b in D(T) that is not in M
      and TreeNode(c) is a child of TreeNode(b) then
   /* downward expansion */
   for each child of TreeNode(b) do
      if (child.in-model = apex) or (child.in-model = apex/frontier) then
         /* downward collision: merge colliding models */
            . . .
         child.in-model := interior
      else
         child.in-model := frontier
      fi
   od
   TreeNode(b).in-model := interior;
   /* expand M */
      . . .
  fi

  if there exists c, a successor of b in D(T) that is not in M
      and TreeNode(c) is the parent or a sibling of TreeNode(b) then
   /* upward expansion */
   if (parent(TreeNode(b)).in-model = frontier) or (parent(TreeNode(b)).in-model = apex/frontier) then
      /* upward collision: merge colliding models */
         . . .
      parent(TreeNode(b)).in-model := interior
   else
      parent(TreeNode(b)).in-model := apex
   fi
   for each sibling of TreeNode(b) do
      if (sibling.in-model = apex) or (sibling.in-model = apex/frontier) then
         /* downward collision: merge colliding models */
            . . .
         child.in-model := interior
      else
         child.in-model := frontier
      fi
   od
   TreeNode(b).in-model := interior;
   /* expand M */
      . . .
  fi
end
```

Figure 3-9: *Algorithm to Expand Model and Detect Collisions*

```
atomic procedure clear-model-field ( node: semantic tree node );
begin
   if (node.in-model $\neq$ frontier) then
      for each child c of node do
         clear-model-field(c)
      od
   fi
   node.in-model := not-in-model
end
```

Figure 3-10: *Clearing in-model Field when Evaluation Process Terminates*



(a): *After expansion of M_2 at X_0 by production $X_0 \to X_1 \cdots X_n$*

(b): *Dependencies across colliding models*

Figure 3-11: *A Downward Collision due to a Downward Expansion*

Model M_2 contains those attributes of X_i that have not yet been evaluated, as well as superior transitive edges among them. To merge M_1^+ and M_2, all that may seem necessary is to delete the subordinate and superior transitive edges among attributes of X_i from M_1^+ and M_2 respectively, and then unite the two resulting graphs, where the union operation removes identical vertices and edges.

The problem with this simple merging approach arises when there is an attribute *a* within the backbone of one of the colliding models, for instance, M_1, that depends on an attribute b in the other colliding model, M_2, and a has already been evaluated and removed from M_1. The attribute a may be affected, through b, by the subtree replacement corresponding to the model M_2. But with the simple merging scheme, a will not be evaluated again since no future expansions of the merged model will reinsert a. (Recall that a model represents the dependencies of a connected region of the semantic tree, and thus is only expanded from the apex and frontier nodes.) This is clearly wrong — the semantic tree will still have inconsistent attributes after all evaluations terminate.

We solve this problem by reinserting all such attributes back into the model, together with the direct dependency edges needed to link them up to attributes still in the model. This is accomplished by the atomic procedure *freeze*, shown in figure 3-12. *Freeze* performs a search of the dependency graph within the backbone of a model, looking for dependent attributes that should be reinserted into the model. However, the search space can be reduced because of the following observation. Referring again to the case illustrated in figure 3-11, the subtree replacement corresponding to the model M_1 can only affect attributes in M_2 through the inherited attributes associated with the node X_i. Conversely, the subtree replacement corresponding to M_2 can only affect attributes in M_1 through the synthesized attributes associated with the node X_0.[11] Thus, for the example of figure 3-11, *freeze* is called for each synthesized attribute of X_0 in model M_1, and for each inherited attribute of X_i in model M_2.

Freeze takes three arguments: (1) an attribute instance, *ai*, (2) a model M, and (3) a worklist S of attribute instances in M that are ready to be evaluated. *Freeze* performs a depth-first traversal of the dependency graph in the region of the semantic tree bounded by the backbone of M, starting from *ai*. Traversal along a path from *ai* stops at an attribute x if any of the following three conditions hold:

1. Attribute x is in the model M (lines [1]-[4] in figure 3-12). In this case, we have gone as far as we need to go along this path.

[11]The proof of this observation follows directly from the definition of "synthesized" and "inherited".

```
       atomic procedure freeze ( ai: attribute instance; M: a model, S: set of
                                                    attribute instances );
       declare
          adjacent_attrs: set of attribute instances;
          b: attribute instance;
       begin
[1]       if ai ∈ M then
[2]          if in-degree of ai in M is 0 then
[3]             Remove ai from S
             fi
[4]          return
          fi
[5]       Mark ai visited;
[6]       adjacent_attrs := find_adjacent_attrs(ai);
[7]       for each attribute instance b in adjacent_attrs do
[8]          if b is not visited then
[9]             freeze(b, M)
             fi
[10]         Add ai and red edge (ai, b) to M
          od
       end /* of freeze */

       function find_adjacent_attrs ( ai: attribute instance )
                         returns set of attributes instances;
       declare
          adjacent_attrs: set of attribute instances;
       begin
[1]       if (TreeNode(ai).in-model = apex/frontier) then
[2]          adjacent_attrs := transitive_transitive(ai)
[3]       else if (TreeNode(ai).in-model = apex) then
[4]          adjacent_attrs := transitive_direct(ai)
[5]       else if (TreeNode(ai).in-model = frontier) then
[6]          adjacent_attrs := direct_transitive(ai)
[7]       else /* TreeNode(ai) is an interior node */
[8]          adjacent_attrs := direct_direct(ai)
          fi
[9]       return(adjacent_attrs);
       end /* of find_adjacent_attrs */
```

Figure 3-12: *Search for Attributes to be Reinserted into a Colliding Model*

2. Attribute x has already been visited (line [8]). In this case, there is no need to continue past x since the rest of the path has been traversed already.

3. Attribute x is on the boundary of M. In this case, continuing along this path leads outside the backbone of M. (When this condition occurs, the value of *adjacent_attrs* is the empty set, for reasons which become clear in the next paragraph, causing the invocation of *freeze* with x to terminate.)

The procedure *find_adjacent_attrs*, also shown in figure 3-12, computes the set of attribute instances that are immediate successors of *ai* within the region of the semantic tree bounded by the backbone of M. An *immediate successor* of *ai* is an attribute b such that the edge (*ai*, b) is in the model, or was in the model at some point but has been removed. Determining the immediate successors of *ai* depends on whether *ai* is at the apex, frontier, or interior of model M's backbone: In the case of attributes at the apex or frontier, M contains (or contained) transitive dependency edges as well as direct dependency edges, whereas for attributes at an interior node, only direct dependency edges are (or were) in the model.

The four functions that are invoked in *find_adjacent_attributes* to determine the immediate successors of an attribute *ai*, depending on the position of *ai* in the model's backbone, are defined as follows:

- *transitive_transitive*: Returns attributes that are successors of *ai* because of a transitive dependency superior to the semantic tree node that *ai* is associated with, or because of a transitive dependency subordinate to the semantic tree node of *ai*.

- *transitive_direct*: Returns attributes that are successors of *ai* because of a transitive dependency superior to the semantic tree node that *ai* is associated with, or because of a direct dependency subordinate to the semantic tree node of *ai*.

- *direct_transitive*: Returns attributes that are successors of *ai* because of a direct dependency superior to the semantic tree node of *ai*, or because of a transitive dependency subordinate to the semantic tree node of *ai*.

- *direct_direct*: Returns attributes that are successors of *ai* because of a direct dependency superior to, or subordinate to, the semantic tree node of *ai*.

These functions use the characteristic graphs of the semantic tree node that *ai* is associated with, and the direct dependencies of the two production instances that apply at the tree node of *ai*.

After a recursive invocation of *freeze* with attribute instance *b* returns, the attribute *ai* and the edge (*ai*, *b*) are added to the model *M* (line [10]). As each recursive call returns, line [10] reinserts into *M* all attributes reachable from the attribute instance that *freeze* was initially called with that were not in *M*, as well as all edges between them. These edges are colored red to differentiate them from other model edges, which have no color, for reasons to be explained below.

If *freeze* adds an edge (*x*, *y*) to *M*, where *x* is an attribute reinserted by *freeze* and *y* was in *M* before *freeze* was invoked, the attribute *y* may be in the worklist *S*. In this case, *y* must be removed from *S* since it is no longer ready for evaluation (lines [2]-[3]).

The reason for coloring the edges that are reinserted into the model by *freeze* is to avoid reevaluating a reinserted attribute whose predecessor does not change in value. That is, suppose that a whole chain of edges and attributes were reinserted between an attribute *ai* associated with X_0 and a successor *b* of *ai* that was in M_1 at collision. Recall that this means that all attributes along the path between *ai* and *b* have already been in M_1 once, reevaluated, and removed. If an attribute *c* along this chain is reevaluated but its value does not change, none of the attributes between *c* and *b* need to be evaluated again.

Figure 3-13 shows the atomic procedure *antifreeze* that is called when an attribute *c* is evaluated and its value does not change. *Antifreeze* removes an attribute *d* (as well as all its outgoing red edges) from the model *M* if two conditions hold: (1) *d* depends only on *c* in the model *M*, and (2) *d* was inserted in *M* by a previous invocation of *freeze*. Red edge coloring is used to determine which dependent attributes were reinserted by *freeze*: An attribute *d* that depends on *c* was inserted into the model by *freeze* if *d* has outgoing red edges.

Antifreeze may cause an attribute instance that was not inserted into the model by *freeze* (and is therefore not removed by *antifreeze*) to become ready for evaluation because of the removal of incident red edges. Lines [5]-[6] of *antifreeze* insert such attributes into the worklist *S*.

```
        atomic procedure antifreeze ( ai: attribute instance; M: a model; S: set of attribute
                                                            instances );
        declare
           b: attribute instance;
        begin
[1]        for each attribute b such that (ai, b) is a red edge in M do
[2]           Remove edge (ai, b) from M;
[3]           if red in-degree of b in M is 0 then
[4]              if red out-degree of b in M is 0 then
[5]                 if in-degree of b in M is 0 then
[6]                    Insert b into S fi
                 else
[7]                 antifreeze(b, M, S)
                 fi
              fi
           od
        end /* of antifreeze */
```

Figure 3-13: *Antifreeze Algorithm*

3.7.1. Merging Models after Expansion Collision

Figure 3-14 gives the complete algorithm for expanding a model by a downward expansion, detecting collisions during expansion, and merging the colliding models. When a collision at a child c of the node n being expanded is detected, the following actions are performed. We denote the model being expanded at node n by M_n, and the model that covers node c by M_c.

1. *Freeze* is called for each inherited attribute of c to reinsert attributes in model M_c that depend on attributes in M_n.

2. The worklists of the two colliding models are combined.

3. The evaluation process using the model at the child node c is terminated. After the collision, the evaluation process that performed the expansion at node n continues with the merged model. This, in effect, combines the two evaluation processes that had been executing independently prior to the collision.

```
atomic procedure asynch-expand ( M: a directed graph; b: an attribute instance; S: set of
                                 attribute instances );
  declare
    c: an attribute instance;
    ExpandedSubordinate: directed graph;
    collision: boolean;
  begin
    if there exists c, a successor of b in D(T) that is not in M
            and TreeNode(c) is a child of TreeNode(b) then
        /* downward expansion */
        ExpandedSubordinate := direct dependencies of production that applies at TreeNode(b);
        for each child of TreeNode(b) do
            if (child.in-model = apex) or (child.in-model = apex/frontier) then
                collision := true;
                for each inherited attribute i of child do
                    freeze(i, model of child, worklist of model of child);
                    S := S ∪ worklist of model of child;
                    Terminate evaluation process using model of child
                od
                child.in-model := interior;
                ExpandedSubordinate := ExpandedSubordinate ∪ (model of child − child.C̄)
            else
                child.in-model := frontier;
                ExpandedSubordinate := ExpandedSubordinate ∪ child.C
            fi
        od
        if (collision) then
            for each synthesized attribute s of TreeNode(b) do
                freeze(s, M, S)
            od
        fi
        TreeNode(b).in-model := interior;
        /* expand M */
        M := (M − TreeNode(b).C) ∪ ExpandedSubordinate;
        Insert into S all vertices of ExpandedSubordinate
                        whose in-degree in M is 0
    fi

    /* upward expansion */
        . . .

  end
```

Figure 3-14: *Algorithm to Expand Model, Detect Collisions, and Merge Models*

4. *Freeze* is called on the synthesized attributes of n to reinsert attributes in model M_n that depend on attributes in M_c.

5. The models M_n and M_c are united by adding to M_n (a) the direct dependencies in the production that applies at n, (b) the model M_c with the superior transitive dependency edges of node c deleted, and (c) the subordinate transitive dependency edges of other children of n.

There are two operations in *asynch-expand* that require further explanation: (1) the union of the model at the colliding child node with the expanding model, and (2) locating the model that covers the colliding child node, and its associated worklist.

The union operation can be performed efficiently (in unit time) if the model is represented as follows. Each attribute instance in the semantic tree has a field listing the attributes adjacent to it in the model; this field is null if the attribute instance is not in the model. The data structure representing the model consists only of a pointer to the semantic tree node that is the apex of the model's backbone; all the other information about the model is maintained in the semantic tree data structure. Therefore, uniting two colliding models during an expansion at node n involves updating the model field of attribute instances in the production that applies at n. For a given grammar, the cost of the union operation is bounded by a constant — the maximum number of attribute instances in a production of the grammar.

A simple solution for locating the model of a node involved in a collision consists of keeping back pointers from the apex node of each model to the model data structure. For a downward collision, this solution takes constant time since the colliding node is the model's apex. However, in the case of an upward collision (which can occur at any node on the frontier of the model), or an initialization collision (which can occur at any node in the model), the semantic tree may have to be traversed from the colliding node to the apex of the model. Thus, the simple solution is not satisfactory.

Our solution for locating the model of a node involved in a collision has constant expected cost for downward and upward collisions, as well as initialization collisions. A unique timestamp is associated with each subtree replacement. The timestamp is employed by the evaluation process to label semantic tree nodes that are in the backbone of the associated model. The model (*i.e.*, a pointer to the model's apex) and worklist of each independent evaluation process are stored in a hash table, and the timestamp of the corresponding evaluation process is used as the key for the hash table. To lookup the model of an arbitrary node in the semantic tree (that is covered by some model), the node's timestamp is used to find the hash table entry containing the node's model.

When two models are merged, the hash table entries for the two models must be updated. Suppose that two models, M_1 and M_2 are merged, and suppose that the apex of M_1 is an ancestor of the apex of M_2 in the semantic tree. We call M_1 the *superior model*, and M_2 the *subordinate model.*

The apex of the merged model is the apex of the superior model. Therefore, the hash table entry for the superior model is not changed by the merge operation, since the model already points to the right node. The hash table entry for the subordinate model, however, must be changed to point to the apex of the superior model. The combined worklist of the merged model is stored in both hash table entries for the two colliding models. With this scheme, even though semantic tree nodes in the merged model are labeled with different timestamps, the correct merged model and worklist will be located for each node in the merged model's backbone.

3.7.2. Merging Models after Initialization Collision

After a subtree replacement at a node r, the initial model created for the new evaluation may overlap an existing model of a previous evaluation in progress, causing a collision. Collisions at model initialization are simpler than those at model expansion because none of the attributes in the new model have been evaluated. Therefore, it is only necessary to call *freeze* on the synthesized attributes of r.

There is however an additional complexity in the initialization case: the existing model involved in a collision may cover the deleted subtree. Thus, the part of the existing model in the replaced subtree must be deleted, and attribute instances in the worklist S associated with nodes in the replaced subtree must be removed. These cleanup operations are performed by the routine *replace-subtree* before the replaced subtree is pruned, as shown in figure 3-15. The part of the model covering node *r* is not deleted since pruning the subtree rooted at *r* does not delete node *r* from the tree. Therefore, the *in-model* field of *r* is set to *frontier* if an initialization collision occurred.

```
atomic procedure replace-subtree ( r: nonterminal node at root of replaced subtree;
                                   S: new subtree );
begin
   if (r.in-model ≠ not-in-model) then
      cleanup-model(r, worklist of model of r);
      if (r.in-model = apex) then
         r.in-model := apex/frontier
      else
         r.in-model := frontier
      fi
   fi

   /* prune subtree rooted at r, and graft subtree S onto r */
      . . .
end /* of replace-subtree */

procedure cleanup-model ( node: semantic tree node; S: set of attribute instances );
begin
   if (node.in-model ≠ frontier) or (node.in-model ≠ apex/frontier) then
      for each child c of node do
         cleanup-model(c)
      od
   fi
   for each attribute instance ai of node do
      if in-degree of ai in model is 0 then
         Remove ai from S
      fi
      ai.model := null;
   od
end /* of cleanup-model */
```

Figure 3-15: *Replace Subtree Operation*

The complete *startup* procedure, which initializes an evaluation process for a new subtree replacement, is shown in figure 3-16. An initialization collision is detected if the root *r* of the replaced subtree is on the frontier of an existing model. If a collision is detected, *freeze* is called for each synthesized attribute of *r* to reinsert attributes in the existing model that depend on attributes in the new subtree at *r*. Uniting the initial model with the existing model is performed by adding subordinate characteristic graph of *r* to the existing model. Since the initial model has been merged with an existing model, no evaluation process has to be initiated for a subtree replacement that causes an initialization collision.

```
atomic procedure startup ( T: semantic tree; r: node of T at root of replaced subtree );
declare
    M: a directed graph;
    S, NeedToBeEvaluated: sets of attribute instances;
begin
    if (r.in-model = apex/frontier) or (r.in-model = frontier) then
        /* initialization collision */
        for each synthesized attribute s of r do
            freeze(s, model of r, worklist of model of r)
        od
        model of r := model of r ∪ r.C;
        Insert into worklist of model of r vertices of r with in-degree 0;
        Insert into NeedToBeEvaluated of model of r the vertices of r
    else
        M := r.C ∪ r.C̄ ;
        S := the set of vertices of M with in-degree 0 in M;
        NeedToBeEvaluated := the set of vertices of M;
        fork asynch-propagate(M, S, NeedToBeEvaluated)
    fi
end
```

Figure 3-16: *Startup Algorithm (revisited)*

3.8. Analysis of Collision-Merging Algorithm

The number of attributes evaluated by the Collision-Merging algorithm for k asynchronous subtree replacements is optimal when initialization collisions occur for each subtree replacement. If models for new evaluation processes do not collide immediately with existing models, however, attributes that are affected by more than one edit may be evaluated unnecessarily.

The bookkeeping costs of the algorithm are analyzed in two parts: (1) the cost of detecting collisions, and (2) the cost of merging the models if there is a collision. Collision detection is performed once when the model is initialized, and each time the model is expanded. Detecting collisions costs $O(1)$, since the algorithm simply checks if any of the semantic tree nodes added by the expansion, or during initialization of a new model, are in another model's backbone. The field of the semantic tree node indicating whether the node is within the backbone of a model can be updated in unit time at each expansion. This field is cleared when the evaluation of the k subtree replacements (all of which have collided) terminates, at a maximum cost of $O(\sum_{i=1}^{k} |AFFECTED_i|)$. (Termination is assumed only for analysis purposes. The Collision-Merging algorithm works correctly even if there is a continuous stream of asynchronous edits.)

The cost of merging two models involved in a collision can be broken into the following items:

1. *Linking the two models at the collision point node*: This operation involves the attributes associated with the collision point node, which is bounded by a constant for a given AG. Thus, this operation costs $O(1)$.

2. *Combining the worklists associated with the two models*: This is a set union operation. Since the two models before collision cover different parts of the semantic tree, the two sets are disjoint. Therefore, there is no need to check for duplicates in the sets being united, and this operation is also $O(1)$.

3. *Deleting the part of a model that covers a replaced subtree in the case of an initialization collision*: The cost of this operation is proportional to the number of attributes in the region of the replaced subtree covered by the

model. The cost can be ''charged'' to the subtree replacement that caused the model to expand over the replaced subtree. Since each distinct part of a model can only be deleted once, the total cost of deleting parts of the model because of subsequent subtree replacements that is charged to each subtree replacement is at most equal to the number of attributes affected by the subtree replacement.

4. *Reinserting attributes in one model that may be affected by attributes in the other model*: This is the algorithm that causes the worst-case complexity. The worst case happens when the following conditions hold:

    - The intersection between the sets $AFFECTED_i$ and $AFFECTED_{i+1}$, $1 \le i \le k-1$, is very small.

    - Model M_i collides with model M_{i+1} at the point when both have performed all their expansions.

    - All but a few of the attributes in $AFFECTED_i$ and $AFFECTED_{i+1}$ have been evaluated at the time of the collision.

    This means that the collision will reinsert ($AFFECTED_i$ + $AFFECTED_{i+1}$) attributes into the merged model, which will eventually be removed without ever being evaluated.

    For $k-1$ collisions between M_i and M_{i+1}, $1 \le i \le k-1$, the maximum cost of inserting evaluated attributes back into the merged model is:

    $AFFECTED_1 + AFFECTED_2 +$ [Collision 1]
    $AFFECTED_1 + AFFECTED_2 + AFFECTED_3 +$ [Collision 2]
    . . .
    $AFFECTED_1 + AFFECTED_2 + \ \ldots\ + AFFECTED_k$ [Collision k-1]

    The worst-case cost of this step of the merging algorithm is $O(\sum_{i=1}^{k} |AFFECTED_i| \cdot i)$. The cost for removing the inserted attributes from the merged model if they do not need to be evaluated is the same.

Thus, the worst-case bookkeeping costs for k colliding models is $O(\sum_{i=1}^{k} |AFFECTED_i| \cdot i)$. We expect that in practice the Collision-Merging algorithm will perform considerably better than the worst case. If k (the number of asynchronous subtree replacements whose models collided) is large, then collisions occur soon after model initialization, and therefore the worst-case scenario analyzed for the cost of merging the colliding models does not arise. If k is very small compared to the average size of $AFFECTED_i$, then the bookkeeping costs are $O(\sum_{i=1}^{k} |AFFECTED_i|)$, which is close to the size of the union of the affected sets, resulting in almost optimal bookkeeping costs.

The algorithms we have described can be extended to minimize the number of attribute evaluations. This may be desirable if the cost of attribute evaluation is much higher than the cost of traversing the semantic tree. In order to do this, we force a collision between the initial model of a new subtree replacement with the existing model as soon as the new subtree replacement occurs. Forcing a collision is done by expanding one or both models, depending on whether one model is an ancestor of the other or not. The model edges added by the expansions required to force a collision are colored red for the same reason that edges added by the *freeze* procedure are colored: to avoid evaluating attributes that are not affected by either subtree replacement. This version of the Collision-Merging algorithm minimizes the number of attributes evaluated, but the worst-case bookkeeping costs are proportional to the size of the semantic tree.

3.9. Related Work

Kaplan and Kaiser were the first to describe an attribute evaluation algorithm to handle multiple asynchronous edits for the general class of noncircular AGs [Kaplan 86]. Their algorithm was later expanded for parallel evaluation on a centralized or decentralized tree in response to multiple asynchronous edits, with multiple editing cursors protected from each other by *firewalls* [Kaiser 90]. Firewalls prevent a subtree replacement from being performed in a region of the tree where an evaluation process is in progress. Their initial work left open the problems addressed in this chapter. Their updated algorithms assume the collision detection and merging algorithms just described. They do not address the problem of maintaining correct characteristic graphs at derivation tree nodes when there are multiple cursors. The latter problem is not solved by firewalls since the characteristic graphs may need to be updated even if a subtree replacement occurs when there are no other evaluation processes in progress.

Neither of the other two incremental attribute evaluation algorithms for handling multiple asynchronous subtree replacements that have been reported are applicable to the general class of noncircular AGs. Geitz gives an algorithm that minimizes the number of attributes reevaluated by maintaining transitive dependency edges between attributes in one model that depend on attributes in another model [Geitz 87]. This algorithm is only applicable to a subset of the partitioned grammars for which the transitive dependency information required by the algorithm can be computed statically from the grammar. Since Geitz does not analyze the complexity of the bookkeeping costs of his algorithm,[12] it is not clear how his algorithm compares with the version of our Collision-Merging algorithm that minimizes attribute reevaluation.

The evaluation strategy in Geitz' algorithm is dynamic while the other reported algorithm for multiple subtree replacements, by Peckham, is static [Peckham 90]. Peckham's work is reviewed in the related work section of the next chapter, which deals with static evaluation algorithms.

Reps *et al.* developed an algorithm for synchronous subtree replacements [Reps 86], applicable to the same subset of the partitioned AGs as Geitz' algorithm. This paradigm was invented in the context of single user environments to allow commands that do not map nicely to single subtree replacements. This algorithm is not effective when changes occur asynchronously: although one can postpone all evaluation until some predetermined number of changes have been made and then apply the synchronous algorithm, the rapid feedback about errors is then lost.

Another algorithm by Reps [Reps 88], also not suitable for multiple users, supports incremental evaluation for a single cursor after an arbitrary movement of the single cursor from any point in the tree to any other. This algorithm is an improvement over the naive approach described in section 3.2, reducing the cost of moving the internal cursor from the root of one subtree replacement to another, but it still completes the attribute evaluation process initiated by one subtree replacement before starting the next and thus may evaluate attributes unnecessarily.

In his work on parallel attribute evaluation for tree transformations, Alblas [Alblas 90] describes a merging operation for overlapping models resulting from different transformations that seems similar to the one described in this chapter. However, Alblas describes the merge operations as "a union operation, *i.e.*, identical arcs and vertices in two graphs become one in the resultant graphs". His algorithm is incorrect

[12]We attempted to perform the analysis ourselves but did not succeed due to the complexity of the algorithm.

because as we described in section 3.7, uniting the two graphs is not sufficient to ensure that after the evaluation terminates all attributes have consistent values.

Chapter 4
Static Evaluators for Incremental Attribute Evaluation of Multiple Subtree Replacements

In the previous chapter, we presented a *dynamic* evaluation strategy for reestablishing attribute consistency in a semantic tree after multiple asynchronous subtree replacements. This chapter describes a *static* evaluation algorithm for the multiple update problem for two restricted classes of attribute grammars.

When evaluating the attributes of a semantic tree *T*, any evaluator must follow the partial order of *T*'s attribute dependency graph. Dynamic evaluators maintain the dependency graph of the semantic tree representing the program at run-time. When a change is made to the program, the dependency graph is updated and attribute evaluations are scheduled by dynamically performing a topological sort on the dependency graph. The disadvantages of a dynamic evaluation strategy are twofold. First, most of the work is done at run-time. In an incremental editor, this degrades the response time after an edit. Second, in order to build the dependency graph, large structures must be kept around, resulting in an incredible use of storage.

Static evaluators overcome both these problems; they are more efficient, both in terms of CPU time as well as memory utilization. Static evaluators precompute *plans* that specify the order of evaluation of attributes of each production in the grammar. These plans are created once for each AG during construction of the grammar's evaluator. At run-time, the evaluator determines the order of attribute evaluations using the plans associated with each production instance in *T*. The disadvantage of static evaluators is that not all well-defined attribute grammars can be evaluated by a static evaluation scheme. However, static evaluators can be constructed for a large subclass of AGs, including most of the ones that arise in practice in the programming language application domain [Reps 89a].

We present a new static incremental evaluator that can handle multiple asynchronous subtree replacements. Our algorithm is applicable to *partitioned* attribute grammars,[13] a subclass of the noncircular attribute grammars that will be defined below. However, since the cost of determining the plans of a partitioned grammar is exponential [Waite 84], we expect our algorithm to be used for *ordered* attribute grammars (OAGs) in practice. OAGs are a subclass of the partitioned attribute grammars for which an efficient algorithm for constructing attribution plans is known [Kastens 80]. Also, the same general idea can be used to extend semi-static evaluation strategies, such as the one described in [Kennedy 76], to handle asynchronous subtree replacements.

The new evaluation algorithm presented in this chapter minimizes the number of attributes evaluated: (1) only attributes affected by each modification are evaluated, and (2) an attribute that is affected by more than one subtree replacement still in progress and which has not yet been evaluated in any of them is evaluated once only. In order to accomplish this for partitioned (or ordered) attribute grammars, some run-time checks are required. We define a subclass of OAGs, called the *pairwise ordered attribute grammars* (POAGs), for which this run-time check can be replaced by a table lookup operation, making the evaluator even more efficient.

Section 4.1 gives an overview of static evaluators. An incremental evaluation algorithm for partitioned (and ordered) AGs when asynchronous subtree replacements are allowed is presented in section 4.2. Section 4.3 defines pairwise ordered attribute grammars, and describes algorithms to construct evaluators for these grammars and record information needed during incremental evaluation. We conclude this chapter with a comparison to other relevant work.

[13]This class was defined by Kastens [Kastens 80], but he used the term ''arranged orderly'' to denote a partitioned grammar. The term ''partitioned'' is now widely used to refer to this class.

4.1. Overview of Static Evaluators

A static evaluator uses a strategy that is precomputed at construction-time by a static analysis of the grammar for evaluating the attributes of a semantic tree. The static evaluators discussed in this chapter are *tree-walk* evaluators that do not follow a pre-specified order of visiting the nodes of the tree (*e.g.*, a preorder traversal), but rather, the tree traversal is defined by the attribute dependencies of the grammar. A tree-walk evaluator "roams" over the semantic tree: upon reaching a node, it evaluates some of the node's attributes, visits some children, returns to the node, evaluates more attributes, and so on, until it finally leaves the node.

A static tree-walk evaluator is guided by *plans*, which are constructed for each production in the grammar. The plan for a production p: $X_0 \rightarrow X_1 \cdots X_n$ is composed of the following basic instructions:

- Eval($X_i.a$) — Evaluates the attribute $X_i.a$ according to the semantic function defining it in production p. $X_i.a$ is a synthesized attribute if $i = 0$ and an inherited attribute if $1 \leq i \leq n$.
- $v(i,k)$ — $\begin{cases} i = 0, & \text{Visits parent of } p \text{ for the } k^{\text{th}} \text{ time.} \\ i > 0, & \text{Visits child } X_i \text{ for the } k^{\text{th}} \text{ time.} \end{cases}$

The evaluator is a table-driven algorithm; a driver runs under control of a table containing the plans for a specific attribute grammar. To evaluate the attributes of a semantic tree T, the driver executes the instructions in the plans associated with the production instances of T. Execution starts with the first instruction of the plan for the root production of T. When an *Eval* instruction is encountered, the specified attribute is evaluated, after which the driver moves on to the next instruction in the same plan. If the instruction is a "visit child" (or "visit parent") instruction, then control is passed to the plan at the child (or the parent) production, resuming execution of that plan where it last left off. The driver halts when the last instruction in the plan of the root production, a "visit parent" instruction, is encountered.

The plan generated for a production p is used to evaluate the attributes defined in an instance of p in <u>any</u> derivation tree of the attribute grammar, independent of p's context in the tree. This restricts the class of AGs for which such static evaluators can be constructed to the *partitioned* attribute grammars. An attribute grammar is partitioned if

> ... for each symbol a partial order over the associated attributes can be given, such that in any context of the symbol the attributes are evaluable in an order which includes that partial order.
>
> [Kastens 80]

To see why this restriction is necessary in order for the plan of a production to be applicable in any context, consider two productions $p: Y \rightarrow \alpha X \beta$ and $q: X \rightarrow \chi$. The plans for p and q cooperate to evaluate the attributes of a node X in the tree. The inherited attributes of X are evaluated by instructions in the plan for p while the synthesized attributes of X are evaluated by instructions in q's plan. Each time control is transferred from production p to production q, the evaluator expects that a particular subset of the synthesized attributes of X will be evaluated before control returns to p. This subset must be evaluated by the plan for every production of the form $X \rightarrow \chi$. The same is true for the inherited attributes of X.

Thus, for each symbol X in the attribute grammar, there must exist a partition dividing the attributes associated with X into disjoint subsets of inherited and synthesized attributes, such that the attributes of X can be evaluated in the (partial) order[14] given by the partition for any occurrence of X in derivation trees of the grammar.

The algorithm for finding a partition for a partitioned attribute grammar, needed to compute the grammar's plans, is exponential, and therefore this class of grammars is of little practical interest. Ordered attribute grammars are a subclass of the partitioned attribute grammars for which there is a polynomial time algorithm for constructing the plans of a static tree-walk evaluator [Kastens 80].[15] Only the algorithm used to construct the plans distinguishes the evaluators for partitioned and ordered AGs; the evaluation driver is the same for both classes. The results described in this chapter are applicable to both partitioned and ordered AGs, with the exception of the section on

[14]The attributes in a subset can be evaluated in a different order in different productions.

[15]We do not describe the algorithm for constructing plans for OAGs in this thesis; the algorithm can be found in [Kastens 80] as well as in [Waite 84].

pairwise ordered attribute grammars (section 4.3), a subset of the ordered attribute grammar class.

In the next two subsections we present non-incremental and incremental drivers, which when combined with plans generated for a partitioned or ordered attribute grammar, form a static tree-walk evaluator for that grammar. The non-incremental driver evaluates all the attributes of a semantic tree while the incremental driver evaluates only those attributes affected by a single subtree replacement. These algorithms form the basis of the evaluator for multiple asynchronous subtree replacements to be described in section 4.2.

4.1.1. Non-Incremental Driver for Static Evaluator

In this subsection and the rest of this chapter, we assume that the nodes of a semantic tree are represented by data structures containing fields for the attributes of that node, pointers to the (direct) descendents and parent of that node, and a field indicating the production applied to that node. For a treenode X, an attribute a of X is referenced by $X.a$, the parent of X by $X.parent$, the i^{th} descendent by $X.child[i]$, and the production applied to X by $X.production$.

The plans constructed for the attribute grammar are collected into a table, *Plan*. $Plan[p]$ contains the plan for production p, and $Plan[p][i]$ denotes the i^{th} instruction in p's plan.

Figure 4-1 shows a stack implementation of a non-incremental driver for partitioned and ordered attribute grammars — this driver evaluates all the attributes in a semantic tree. The stack contains pairs of a reference to a treenode (stored in the field *currentNode*) and an index into the plan for that node indicating the next instruction to be executed (stored in the field *planIndex*). The top of the stack is available in the global variable *StackTop*. Thus, *StackTop.currentNode* indicates the node whose plan is currently active (*i.e.*, being executed), and *StackTop.planIndex* indicates the instruction to be executed next by the driver. The stack is initialized with the root node of the tree and the index of the first instruction in the plan of the root node.

The function $MapDown(p,k)$ used in the driver returns the next instruction to be executed in the plan for production $p: X_0 \rightarrow X_1 \cdots X_n$ after the k^{th} visit to X_0. The values returned by *MapDown* are determined during construction of the plans for a specific AG.

```
procedure OAGevaluate(root: root node of semantic tree to be evaluated);
begin
   push(root, MapDown(root.production, 1));
   repeat
      case Plan[StackTop.currentNode.production][StackTop.planIndex] of
         Eval(X.a)  : call semantic function defining X.a;
                      increment(StackTop.planIndex)
         v(i,k), i > 0: /* descendent visit */
                      increment(StackTop.planIndex);
                      push(StackTop.currentNode.child[i],
                           MapDown(currentNode.child[i].production, k))
         v(0,k)     : /* ancestor visit */
                      pop
      esac
   until StackIsEmpty
end
```

Figure 4-1: *Non-Incremental Driver*

The actions of the driver depend on whether the next instruction to be executed is an *Eval* or *visit* instruction. An *Eval* instruction is executed by invoking the semantic function defining the specified attribute. Then the driver continues with the next instruction in the same plan.

If the next instruction is a "visit child" instruction, the driver must resume executing the plan for the production instance at the child node. This is accomplished by pushing the child's plan on top of the stack. The next instruction to be executed in the child's plan is determined by *MapDown*.

If the next instruction is a "visit parent" instruction, then the plan for the parent node must be resumed. This is accomplished by popping the stack, so that the stack element containing the execution information of the parent plan is again at the top of the stack.

The algorithm terminates when the stack is empty, which happens when the last instruction of the plan for the production at the root of the tree (a "visit parent" instruction) is executed.

4.1.2. Incremental Driver for Static Evaluator

Optimal algorithms for performing incremental attribute evaluation in response to a single subtree replacement for ordered attribute grammars have been described by Yeh [Yeh 83] and Reps and Teitelbaum [Reps 89a]. The plans used by these incremental evaluators are identical to the ones used in the non-incremental version, and for OAGs are determined by the algorithm described by Kastens [Kastens 80]. The driver described in this subsection can therefore be used for partitioned attribute grammars as well.

Our presentation of the driver routine for an incremental evaluator that handles single subtree replacements combines the somewhat different approaches used by Yeh [Yeh 83] and Reps and Teitelbaum [Reps 89a]. Suppose a subtree S of a consistently attributed tree T is replaced by another tree, S', which is also consistently attributed. Let T' be the tree T with S replaced by S'. The incremental algorithm described below evaluates the minimum number of attributes in T' to reestablish attribute consistency.

Initially, there are two production instances in T' that may have inconsistent attributes. These are the two productions at the point of subtree replacement. If r is the nonterminal occurrence at the root of S' (and necessarily of S), then the two productions are:

$$p: X_0 \rightarrow X_1 \cdots X_m, \quad where\ r = X_i, \quad 1 \le i \le m, \quad \text{and}$$
$$q: r \rightarrow Y_1 \cdots Y_n$$

The incremental driver starts by executing the first instruction of the plan for production p. Since plans associated with production instances not affected by the subtree replacement do not have to be evaluated, additional information must be maintained to indicate which production instances are affected. This information is stored in the field *Reactivated* of semantic tree nodes deriving production instances that may have affected attributes. Initially, the nodes X_0 and r, which derive the two productions p and q at the point of subtree replacement, are marked *Reactivated.*

The incremental driver routine for ordered and partitioned AGs is given in figure 4-2. It is similar to the non-incremental version described in the previous subsection, except

```
procedure IncOAGevaluate(T: semantic tree; r: node at root of replaced subtree);
declare
   neighborNode: node in semantic tree;
begin
   Mark r and r.parent Reactivated;
   Initialize stack;
   /* start evaluation of plan for production at parent of r */
   push(r.parent, MapDown(r.parent.production, 1));
   forever do
      case Plan[StackTop.currentNode.production][StackTop.planIndex] of
         Eval(X.a)  : call semantic function defining X.a;
                      increment(StackTop.planIndex);
                      if NewValue(X.a) ≠ OldValue(X.a) then
                         if (X = StackTop.currentNode) and (X.a is synthesized) then
                            neighborNode := StackTop.currentNode.parent
                         else if (X is the ith child of StackTop.currentNode) and
                               (X.a is inherited) then
                            neighborNode := StackTop.currentNode.child[i]
                         else
                            neighborNode := Null
                         fi
                         if (neighborNode ≠ Null) and
                               (X.a has successors in neighborNode.production) then
                            Mark neighborNode Reactivated
                         fi
                      fi
         v(i,k), i > 0: /* descendent visit */
                      increment(StackTop.planIndex);
                      if StackTop.currentNode.child[i] is marked Reactivated then
                         push(StackTop.currentNode.child[i],
                            MapDown(StackTop.currentNode.child[i].production, k))
                      fi
         v(0,k)     : /* ancestor visit */
                      if Plan[StackTop.currentNode.production][StackTop.planindex]
                            is the last instruction in the plan then
                         Mark StackTop.currentNode not Reactivated
                      fi
                      if StackTop.currentNode = root of T then return fi
                      if StackTop.currentNode.parent is marked Reactivated then
                         pop
                      else if Plan[StackTop.currentNode.production][StackTop.planindex]
                            is the last instruction in the plan then
                         return
                      else
                         increment(StackTop.planIndex);
                      fi
      esac
   od
end
```

Figure 4-2: *Incremental Driver*

that the *Reactivated* field is used to limit the scope of attribute evaluations to only those affected. When an attribute a is evaluated, if its value changes and it is an argument in a semantic function defining another attribute b, then the treenode deriving the production instance where b is defined is marked *Reactivated.* "Visit child" and "visit parent" instruction are skipped if the child or the parent are not marked *Reactivated.* Otherwise, they are executed in the same way as in the non-incremental algorithm. The *Reactivated* field of a semantic tree node is cleared when the last instruction in the node's plan (a "visit parent" instruction) is executed.

The driver halts when a "visit parent" instruction in the plan for the root of the semantic tree is encountered, or when the last instruction in a plan is encountered, and the parent is not marked *Reactivated.*

If the incremental driver is implemented using a stack, as we have shown in figure 4-2, the stack must be initialized to be in the same configuration as the stack used by the non-incremental evaluator at the moment that the plan associated with the parent of r is first visited. An algorithm for initializing the stack is given in [Yeh 83]. This initialization cost can be avoided if the recursion implicit in the stack implementation is eliminated, as in the iterative incremental driver of [Reps 89a]. We chose to describe the evaluators in this chapter using a stack implementation because the iterative versions are more complicated. We do not include the cost of initializing the stack in the analysis of the evaluator, however, since this cost can be eliminated.

4.2. Static Incremental Evaluator for Multiple Asynchronous Subtree Replacements

The key idea behind the static evaluation algorithm for k asynchronous subtree replacements is that the visit sequences followed by the k evaluation processes are *interleaved* in such a way that an attribute affected by more than one subtree replacement, and not yet evaluated, is evaluated at most once. The plans used by this evaluator are the same as for the non-incremental and incremental/single-subtree-replacement evaluators described in the preceding section. In this section we present a new incremental driver routine for the multiple subtree case.

The driver consists of three procedures: *StartUp*, *Schedule*, and *Evaluate*, shown in figures 4-3, 4-4, and 4-5, respectively. *Startup* is invoked when a subtree replacement occurs, possibly interrupting another evaluation in progress. *Schedule* determines the interleaving order of multiple evaluation processes in progress, and is invoked by *Startup* and *Evaluate* as we explain below. *Evaluate* is similar to the driver routines of the preceding section — it executes the instructions in (affected) plans of a semantic tree. It differs in its actions for skipped visits to a parent or child node, where by calling the *Schedule* routine, it maintains the correct interleaving order of the multiple evaluation processes in progress.

We now describe the details of each of these algorithms, using a stack implementation of the evaluator as before. However, in this algorithm, a separate stack is needed for each evaluation process initiated by a different subtree replacement in progress. The global variable *StackTop* now indicates the top of the stack for the evaluation process that currently has control of the evaluator — the one whose affected plans are being executed. We call the evaluation process that has control of the evaluator the *active* evaluation process.

Other evaluation processes initiated by asynchronous subtree replacements wait for their turn to take control of the evaluator. The interleaving order for evaluating the plans of different evaluation processes is determined from the *computation sequence* of the semantic tree. The computation sequence of a semantic tree T is a linearization of the plans associated with the production instances of T, achieved by simulating the operation of an evaluator on T, where instead of executing the instructions, they are (conceptually) appended to the computation sequence [Engelfriet 82]. Note that the computation sequence is a dynamic property of a semantic tree — it changes whenever the semantic tree is modified.

A data structure, *PendingList*, records the state of evaluation processes waiting to take control of the evaluator. The state of a blocked evaluation process consists of a reference to the top of the stack used by the evaluation process. Thus, the i^{th} element in the pending list, *PendingList*[i], points to the top of the stack of the i^{th} pending evaluation process.

The pending list is ordered, with the evaluation that will be resumed first at the head of the list. The ordering of this list is determined according to the computation sequence of the semantic tree. Consider two evaluation processes E_1 and E_2 whose state is saved in the pending list. Let $NextInst_1$ and $NextInst_2$ denote the next instructions to be executed for each of these evaluation processes once they have control of the evaluator.[16] Then, E_1 and E_2 are saved in the i^{th} and j^{th} position of *PendingList*, where $i < j$, if $NextInst_1$ comes before $NextInst_2$ in the computation sequence of the semantic tree.[17] The use of the computation sequence to order the pending evaluations is the key to achieving the second optimality requirement stated in chapter 3, section 3.1:

> For any $k > 1$ modifications affecting the same attribute a, where the k evaluation processes are still in progress and none have yet evaluated a, the algorithm will evaluate a at most once.

We assume that the *Evaluate* routine receives an interrupt signal when the user issues an editor command for performing a subtree replacement. The notation ''On Interrupt'' is used to define the sequence of statements that are executed when an interrupt is received. When interrupted, *Evaluate* finishes evaluating the current instruction, and then suspends the evaluation process that was currently executing, saves the suspended process's state on the pending list, and returns.

The semantic tree is then modified according to the subtree replacement command issued by the user, and procedure *StartUp* is invoked. *StartUp* initializes the stack and marks the set of nodes with affected attributes for the new evaluation. The stack of the new evaluation is stored in the global variable *StackTop*, making the new evaluation (temporarily) the active one. That is, if the evaluator is resumed (by a call to *Evaluate*), it would start executing the plans affected by the new subtree replacement. But this would be wrong if the new evaluation process should be executed after other blocked evaluations in the pending list according to the computation sequence of the tree.

---

[16]Recall that the next instruction for an evaluation process whose top of stack is *StackTop* is available in *Plan*[*StackTop.currentNode.production*][*StackTop.planIndex*], and the top of the stack used by the i^{th} evaluation process in *PendingList* is *PendingList*[i].

[17]An algorithm for comparing the position of two instructions in the computation sequence for partitioned attribute grammars is described below in subsection 4.2.1. A more efficient algorithm for pairwise ordered attribute grammars is given in section 4.3.

```
global variables
    T: semantic tree;
    StackTop: top of stack of currently active evaluation process;
    PendingList: list of evaluations waiting to be restarted, ordered according to
                which should be restarted first;

procedure StartUp(r: node at root of replaced subtree);
begin
    Mark r and r.parent Reactivated;
    StackTop := Initialize stack;
    /* evaluation starts with plan for production at parent of r */
    push(r.parent, MapDown(r.parent.production, 1));
    Schedule()
end
```

Figure 4-3: *StartUp Algorithm*

Therefore, *Schedule* is called first to ensure that the new evaluation is processed in the correct order.

Schedule determines which one of two evaluation processes — the currently active one whose state is available in the global variable *StackTop*, or the first one on the pending list — should be the active evaluation. It compares the position of the next instruction to be executed in these two evaluations processes in the computation sequence of the tree. If the next instruction in the currently active evaluation process is first in the computation sequence, then it remains the current evaluation. Otherwise, the current evaluation process is suspended and inserted into the pending list, and the first evaluation on the pending list is activated. *Schedule* then calls *Evaluate*, which starts (or continues) executing the instructions of the current evaluation process.

The rationale behind the operation of the scheduler is that the evaluation that is resumed will eventually "reach" the other evaluation that was placed on the pending list. This reasoning may be incorrect if a visit to the child or parent that would have reached the other evaluation is skipped because the child or parent were not marked *Reactivated*. Therefore *Evaluate* must handle skipped visits in a special way: If a "visit child" or "visit parent" instruction is about to be skipped because the child or parent node is not marked *Reactivated*, *Schedule* is called.

```
procedure Schedule();
declare
    TempStackTop: temporary variable for top of stack;
begin
    if Plan[StackTop.currentNode.production][StackTop.planIndex] is after
        Plan[PendingList[1].currentNode.production][PendingList[1].planIndex]
        in computation sequence of T then
            /* Swap currently active evaluation with first element in PendingList */
            TempStackTop := PendingList[1];
            Insert StackTop in appropriate place in PendingList;
            StackTop := TempStackTop;
    fi
    Evaluate()
end
```

Figure 4-4: *Schedule Algorithm*

The currently active evaluation process "reaches" a pending evaluation when the instruction that it is about to execute is the pending evaluation's next instruction. This is checked by the *CheckIfReachedPendingEval* routine shown in figure 4-6, which is called from *Evaluate* before executing each instruction. When a pending evaluation is reached, it is removed from *PendingList* — the pending evaluation is subsumed by the current evaluation.

One last issue that needs to be addressed to complete the static evaluation algorithm for multiple asynchronous subtree replacements is the following: *How does replacing a subtree affect pending evaluations*? In other words, what happens to a pending evaluation whose next plan instruction is associated with a node in the subtree to be replaced? Or, what about a pending evaluation that requires multiple visits to the subtree to be replaced, only some of which were completed before the subtree was replaced?

In the following discussion, let S denote the subtree to be replaced and r the root node of S, where the two productions that apply at r are:

$$p: X_0 \rightarrow X_1 \cdots X_m, \quad \textit{where } r = X_i, \quad 1 \le i \le m, \quad \text{and}$$
$$q: r \rightarrow Y_1 \cdots Y_n$$

We refer to the pending evaluation process by $E_{pending}$, and the evaluation process that will be initiated when the subtree S is replaced by E_{new}.

```
procedure Evaluate();
declare
    neighborNode: node in semantic tree;
begin
    forever do
        CheckIfReachedPendingEval();
        case Plans[StackTop.currentNode.production][StackTop.planIndex] of
            Eval(X.a)    : call semantic function defining X.a;
                           increment(StackTop.planIndex);
                           if NewValue(X.a) ≠ OldValue(X.a) then
                               if (X = StackTop.currentNode) and (X.a is synthesized) then
                                   neighborNode := StackTop.currentNode.parent
                               else if (X is the i^th child of StackTop.currentNode) and
                                       (X.a is inherited) then
                                   neighborNode := StackTop.currentNode.child[i]
                               else neighborNode := Null fi
                               if (neighborNode ≠ Null) and
                                       (X.a has successors in neighborNode.production) then
                                   Mark neighborNode Reactivated
                               fi
                           fi
            v(i,k), i > 0 : /* descendent visit */
                           increment(StackTop.planIndex);
                           if StackTop.currentNode.child[i] is marked Reactivated then
                               push(StackTop.currentNode.child[i],
                                   MapDown(StackTop.currentNode.child[i].production, k))
                           else /* skipping this visit — check interleaving order */
                               Schedule()
                           fi
            v(0,k)       : /* ancestor visit */
                           if Plan[StackTop.currentNode.production][StackTop.planindex]
                                   is the last instruction in the plan then
                               Mark StackTop.currentNode not Reactivated
                           fi
                           if StackTop.currentNode = root of T then return fi
                           if StackTop.currentNode.parent is marked Reactivated then
                               pop
                           else if Plan[StackTop.currentNode.production][StackTop.planindex]
                                   is the last instruction in the plan then
                               /* end of currently active evaluation process */
                               /* resume first pending evaluation process, if any */
                               if PendingList is empty then return
                               else
                                   StackTop := PendingList[1];
                                   Delete first item from PendingList
                               fi
                           else /* skipping this visit — check interleaving order */
                               increment(StackTop.planIndex);
                               Schedule()
                           fi
        esac
    od
    on interrupt do
        Insert (StackTop) in first place in PendingList;
        return
    od
end
```

Figure 4-5: *Evaluate Algorithm*

```
procedure CheckIfReachedPendingEval();
begin
  if (StackTop.currentNode = PendingList[1].currentNode) and
        (StackTop.planIndex = PendingList[1].planIndex) then
      Delete first item from PendingList
  fi
end
```

Figure 4-6: *Algorithm to Check if a Pending Evaluation has been Reached*

Recall that the evaluation for a subtree replacement at r starts with the first instruction in plan p. The plan for production p divides the computation sequence of the semantic tree into regions, as depicted in figure 4-7. The instructions in p's plan are numbered $I_{p,1}, I_{p,2}, \ldots, I_{p,n}$, where n is the number of instructions in the plan.

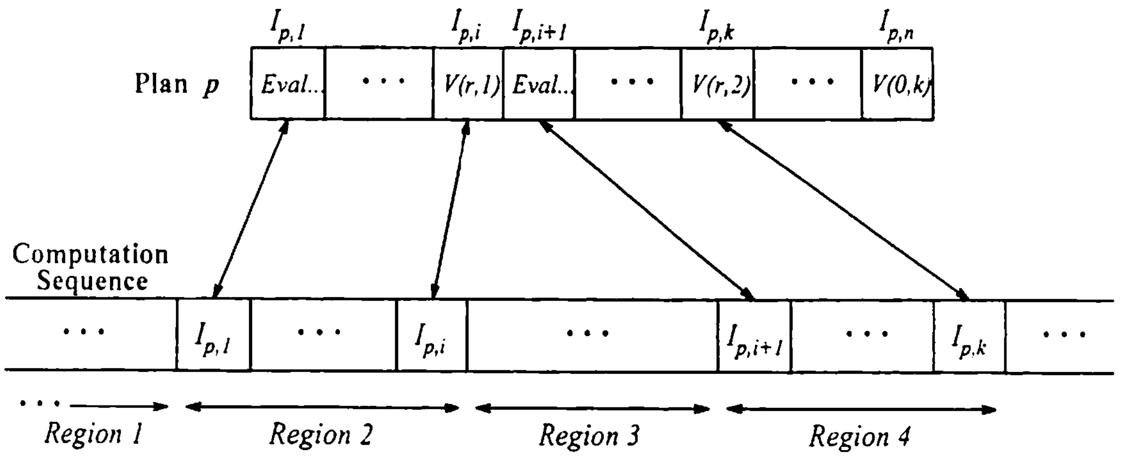


Figure 4-7: *Regions of the Computation Sequence*

The first region consists of the instructions in the computation sequence before the first instruction in p's plan, $I_{p,1}$. Suppose a pending evaluation, $E_{pending}$, would execute an instruction in this region when resumed. Then, the *Startup* and *Schedule* algorithm would place E_{new} after $E_{pending}$ in the pending list. Nothing else is required for handling pending evaluations in this region.

The second region is delimited by the first instruction in p's plan, $I_{p,1}$, and the first "visit child" instruction to node r, instruction $I_{p,i}$. If a pending evaluation $E_{pending}$ is in this region, then E_{new} is scheduled for evaluation first. E_{new} eventually either reaches $E_{pending}$, at which point the two evaluation processes are merged, or skips a visit instruction causing $E_{pending}$ to become the active evaluation. Again, nothing special has to be done for pending evaluations in the second region.

The third region consists of instructions between a "visit child r" instruction in p's plan and the "visit parent" instruction in q's plan that returns control to p; *i.e.*, all instructions executed during one visit to the subtree rooted at r. The next instruction in a pending evaluation in this region is in a plan associated with a node in the subtree S to be replaced. This case requires special handling since replacing the subtree would leave dangling references in the pending list.

A pending evaluation process $E_{pending}$ whose next instruction falls in the third region of the computation sequence is handled by combining it with the evaluation process E_{new}, deleting its corresponding entry from the pending list. We can think of this as rolling back $E_{pending}$ to the "visit child r" instruction at the end of the previous region. This instruction is in the plan associated with the production at the parent of node r, the root of the replaced subtree. E_{new} will eventually reach this instruction since the parent node is marked *Reactivated*, and therefore all the instructions in the parent's plan will be executed (although visits can be skipped). So combining the two evaluations early is correct. Pending evaluations in the third region can be easily identified since the node at the top of the stack of such an evaluation is a descendent of the root of the subtree to be replaced.

Note that combining $E_{pending}$ with E_{new} does not mean that we are reevaluating attributes more than once — any *Eval* instructions executed by $E_{pending}$ within this region prior to the subtree replacement evaluated attributes in the replaced subtree S, and therefore these will not be evaluated again by E_{new}.

The remaining instructions in the computation sequence up to the last "visit child r" instruction in plan p can be divided into region pairs similar to the second and third regions. The fourth (sixth, eighth, *etc.*) region starts with an instruction in p's plan following a "visit child r" instruction and ends with a "visit child r" instruction. The fifth (seventh, ninth, *etc.*) region starts with the first instruction in q's plan executed for this visit to r, and ends with a ''visit parent'' instruction.

Pending evaluations in region four (sixth, *etc.*) are handled like those in the second region, except that any nodes in *Reactivated* that are in the subtree S are removed from the set. Pending evaluations in this region visited the subtree S when they were active, and may have propagated attribute values from the subtree S to other parts of the tree. The following argument shows why this will not result in inconsistent attributes. The propagation of attribute values from S to other parts of the semantic tree must flow through the synthesized attributes of r. If the synthesized attributes of r change as a result of the subtree replacement, then attributes in other parts of the tree that depend on them (and whose current value is based on the replaced subtree) will be reevaluated because they are affected by the subtree replacement. If the synthesized attributes do not change, then those attributes whose value is based on the replaced subtree are still consistent with the new subtree. Thus, all attributes in the semantic tree will be consistent when all evaluation processes have terminated.

Pending evaluations in the fifth (seventh, *etc.*) region are handled like those in the third region. Pending evaluations whose next instruction occurs after the last ''visit child r'' instruction in the computation sequence are not affected by the replacement of subtree S.

4.2.1. Determining Relative Order Among Plan Instructions

The *Schedule* algorithm described above needed to determine whether an instruction i_1 in plan p_1 associated with treenode n_1 occurs before another instruction i_2 in plan p_2 associated with treenode n_2 in the computation sequence of the semantic tree T. This can be done as follows:

(Step 1) Find the least common ancestor (*LCA*) of n_1 and n_2.

(Step 2) Find the next ''visit parent'' instruction following i_1 in plan p_1.

(Step 3) Simulate the operation of the evaluator to determine the instruction that would be resumed in the parent plan.

(Step 4) Repeat steps (2) and (3), each time going up to the parent plan, until instruction j in the plan for *LCA* is encountered. (If n_1 is equal to the *LCA*, then $j = i_1$.)

(Step 5) Repeat steps (2) and (3), but this time for i_2 in plan p_2 until instruction k in the plan for *LCA* is encountered. (If n_2 is equal to the *LCA*, then $k = i_2$.)

If $j < k$, then the answer to the question "*Does instruction i_1 come before instruction i_2 in the computation sequence of T?*" is yes; otherwise the answer is no.

4.2.2. Analysis of Multiple Subtree Replacement Static Evaluator

The static evaluator described in this section minimizes the number of attributes reevaluated for multiple asynchronous subtree replacements. In particular, suppose that a subtree replacement affecting an attribute instance a occurs, but before a is evaluated, a second subtree replacement operation, also affecting the same attribute instance a, is performed. The evaluator shown in figures 4-3, 4-4, and 4-5 evaluates a at most once.

To prove this, suppose there were no further subtree replacements, and let T denote the tree after the second subtree replacement. The evaluator processes instructions in the plans associated with nodes in T in the order defined by the computation sequence of T. Thus, no instruction is executed more than once. In particular, the *Eval* instruction that evaluates the affected attribute instance a is evaluated at most once.

Furthermore, if the value of an attribute a was changed as a result of a subtree replacement, but before any other attributes that depend on a were evaluated, a second subtree replacement was performed, causing a's value to be changed back to what it was originally, the algorithm will not evaluate any of the attributes that depend on a. Although the semantic tree nodes containing attributes dependent on a may have been marked *Reactivated* by the first evaluation process, the new evaluation process will evaluate a before any of a's dependents, and therefore no more nodes are marked *Reactivated* because of the (temporary) value of attribute a.

The cost of interleaving evaluation processes for k subtree replacements to minimize the number of attributes reevaluated is due to calls to the procedure *Schedule*. The non-unit cost operations of *Schedule* are (1) the comparison of the position of two instructions in the computation sequence, and (2) the insertion of an evaluation process in the pending list.

Let n be the number of nodes in the semantic tree. Comparing two instructions in the computation sequence using the algorithm described in the subsection 4.2.1 involves a least common ancestor operation, and a traversal of the tree from the nodes with which the instructions are associated to their least common ancestor node, which costs $O(n)$ in the worst case. Several instructions of the plan for each node on the path to the least common ancestor may have to be examined to find the next ''visit parent'' instruction, but the number of instructions in a plan is a (usually small) constant for a particular attribute grammar. Therefore, the worst-case cost of comparing two instructions is $O(n)$.

Inserting an element in the pending list requires at most k comparisons, each of which costs $O(n)$, with a total cost of $O(k \cdot n)$. Therefore the worst-case cost of one invocation of *Schedule* is $O(k \cdot n)$.

For each subtree replacement, *Schedule* is called once by *Startup*, and each time *Evaluate* is about to skip a visit. The number of skipped visits is at most proportional to the number of attributes affected by the subtree replacement since plan size is a constant of the grammar. Therefore, the total worst-case bookkeeping cost for k subtree replacements is $O(|ASYNC\text{–}AFFECTED| \cdot n \cdot k)$, where *ASYNC–AFFECTED* is the total number of attributes reevaluated as a result of the k subtree replacements.

4.2.3. Improvements

The evaluation algorithm given above evaluates the minimum number of attributes, but its bookkeeping costs can be improved if we can find a more efficient method for determining the relative order among plan instructions, such as precomputing this information at evaluator-construction time. It turns out that this cannot be done for every ordered attribute grammar. An ordered attribute grammar for which it is not possible to precompute the relative order among plan instructions is shown in figure 4-8. Figure 4-9 gives possible attribution plans for the productions in this grammar, such as would be constructed by the algorithm given in [Kastens 80].

```
p1:  α ::= β X γ.              p4:  Y ::= W.
       { X.a = ... ;                  { W.a = Y.a ;
         X.c = X.b ;                    Y.b = W.b ;
         ... ; }                        W.c = Y.c ;
                                        Y.d = W.d ; }

p2:  X ::= Y.                  p5:  W ::= Z.
       { Y.a = X.a ;                  { W.b = W.a ;
         X.b = Y.b ;                    Z.a = W.c ;
         Y.c = X.c ;                    W.d= Z.b ; }
         X.d = Y.d ; }

p3:  Y ::= Z.                  p6:  Z ::= Q.
       { Z.a = Y.a ;                  { Q.a = Z.a ;
         Y.b = Z.b ;                    Z.b= Q.b ; }
         Y.d = Y.c ; }
```

Figure 4-8: *Attribute Grammar that is not Pairwise Ordered*

The reason that we cannot determine at construction time whether instruction i_1 in plan p_1 is executed before instruction i_2 in plan p_2 is that the answer depends on the structure of the tree containing the two productions p and q associated with the plans p_1 and p_2, respectively. Consider the two attributed trees, T_1 and T_2, shown in figure 4-10 below. Production p is $X ::= Y$ and production q is $Z ::= Q$. If the plan for production q is the current one, and instruction "Evaluate $Q.a$" is being executed, then when the plan for p is eventually resumed, the next instruction is "Evaluate $X.b$" in the case of T_1, whereas in the case of T_2, the next instruction is "Evaluate $X.d$".

In the next section, we define a subclass of OAGs, called the pairwise ordered attribute grammars, for which it is possible to precompute the relative order among plan instructions.

Evaluate *Y.a*
Move to *Y*
Evaluate *X.b*
Move to parent
Evaluate *Y.c*
Move to *Y*
Evaluate *X.d*
Move to parent

a) Plan for *X* ::= *Y*

Evaluate *Z.a*
Move to *Z*
Evaluate *Y.b*
Move to parent
Evaluate *Y.d*
Move to parent

b) Plan for *Y* ::= *Z*

Evaluate *W.a*
Move to *W*
Evaluate *Y.b*
Move to parent
Evaluate *W.c*
Move to *W*
Evaluate *Y.d*
Move to parent

c) Plans for *Y* ::= *W*

Evaluate *W.b*
Move to parent
Evaluate *Z.a*
Move to *Z*
Evaluate *W.d*
Move to parent

d) Plan for *W* ::= *Z*

Evaluate *Q.a*
Move to *Q*
Evaluate *Z.b*
Move to parent

e) Plan for *Z* ::= *Q*

Figure 4-9: *Attribution Algorithms for Attribute Grammar of Figure 4-8*

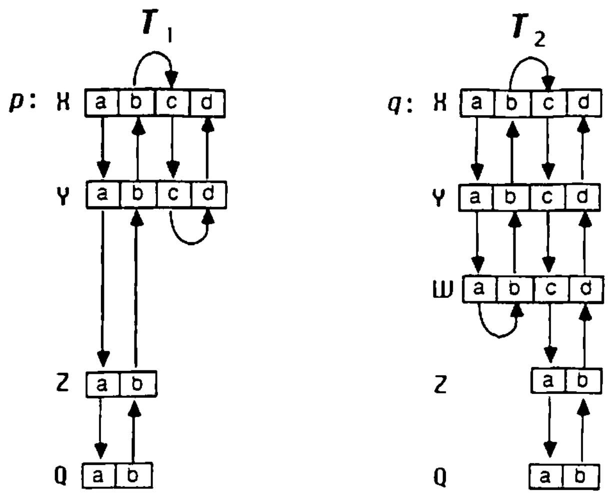


Figure 4-10: *Two Semantic Trees*

4.3. Pairwise Ordered Attribute Grammars

Pairwise ordered attribute grammars are defined as a subclass of ordered attribute grammars.[18] An AG is pairwise ordered if:

1. It is ordered, and
2. For each pair of symbols, X and Y, such that $X \overset{+}{\rightarrow} Y$, a partial order over the attributes of X and Y can be given, such that in any semantic tree where X is an ancestor of Y, the attributes of X and Y are evaluable in an order which includes that partial order.

4.3.1. Algorithm to Compute Plans for POAGs

In this section we describe an algorithm that constructs plans for POAGs according to the definition given above. The algorithm is modeled after Kasten's original algorithm to construct visit-sequences for ordered attribute grammars [Kastens 80]. Only the steps that differ from Kasten's algorithm are described in detail here. Furthermore, we make use of an algorithm to compute transitive dependencies between pairs of symbols in an attribute grammar that was published in [Reps 86]. The details of this algorithm are also not repeated below.

In the algorithm below we use the following notation:

- $A(X)$ is the set of attributes associated with the nonterminal symbol X. $A(X)$ is divided into two disjoint subsets, $I(X)$, containing the inherited attributes of X, and $S(X)$, containing the synthesized attributes of X.
- SF is the set of semantic functions associated with the productions in the grammar. SF_p is the set of semantic functions associated with production p.
- The relation TDS_X contains direct and transitive dependencies between attributes of a nonterminal symbol X.
- The relation TDP_p contains direct and transitive dependencies between attribute occurrences in production p.
- The relation $TDPS_{X,Y}$ contains direct and transitive dependencies between attributes of symbols X and Y, where $X \overset{+}{\rightarrow} Y$.

[18]POAGs are a subclass of ordered attribute grammars and not partitioned attribute grammars because the plans for these grammars are constructed using a modified version of Kastens algorithm for generating the plans for OAGs.

Step 1 and Step 2: Computation of TDS_X and $TDPS_{X,Y}$.

Method: Use algorithms described in the appendix of [Reps 86].[19,20] Note that TDP_p is not computed in these first two steps (as is done in [Kastens 80]) but in step 4. This is done only to simplify the description of the algorithm.

Step 3: Use TDS_X to partition $A(X)$ into subsets $A(X,i)$, $i = 1, \ldots, m$, such that $A(X,i)$ is a subset of $I(X)$ for odd i and a subset of $S(X)$ for even i. The attributes of X can be evaluated in the order $A(X,1), \ldots, A(X,m)$. The output of this step is a vector *PARTITION* describing the disjoint partitions of $A(X)$.

Method: Same as Step 3 of Kasten's algorithm.

Step 4: Computation of TDP_p.

Method: The algorithm is given in figure 4-11. Arcs are added to the (initially empty) TDP_p for the direct dependencies among attribute occurrences in p; the transitive dependencies among attributes of each symbol X in p (given by TDS_X); the transitive dependencies among attributes of the left-hand side symbol X of p and occurrences of each unique symbol Y in the right-hand side of p (given by $TDPS_{X,Y}$); and the dependencies among attributes of each symbol X due to the partitions of X. After adding an edge to TDP_p, other edges required to transitively close TDP are also added. This is accomplished by the function *AddArcTrans* which is the same as defined in [Kastens 80].

If each TDP_p is acyclic, then the AG is a POAG.

Step 5: Construction of visit-sequences.

Method: Same as Step 5 of Kasten's algorithm.

[19]Our notation follows that of Kastens. It differs from the notation used in [Reps 86], where $\overline{DS}(X)$ and $\overline{DP}(X,Y)$ are used instead of TDS_X and $TDPS_{X,Y}$ respectively.

[20]Reps *et al.* use the relation $TDPS_{X,Y}$ in their algorithm to handle multiple *synchronous* subtree replacements.

```
procedure step4();
begin
   for each production $p: X_0 \rightarrow X_1 \cdots X_k$ do
      /* add direct dependencies among attribute occurrences in $p$ */
      for each $f \in SF_p$ defining $X_j.b$ do
         for each argument $X_i.a$ of $f$ do
            if $(X_i.a, X_j.b) \notin TDP_p$ then AddArcTrans($TDP_p$, $(X_i.a, X_j.b)$) fi
         od
      od
      /* add transitive dependencies among attributes of each symbol */
      /* $X$ in $p$ (given by $TDS_X$) */
      for each unique $X_i$ in $p$ do
         for each edge $(c,d)$ in $TDS_{X_i}$ do
            let $(X_i.a, X_i.b) = (c,d)$ in
               for each occurrence $X_i'$ of $X_i$ in $p$ do
                  if $(X_i'.a, X_i'.b) \notin TDP_p$ then AddArcTrans($TDP_p$,$(X_i'.a, X_i'.b)$) fi
               od
            ni
         od
      od
      /* add transitive dependencies among attributes of each pair of */
      /* symbols $X$ and $Y$ in $p$ (given by $TDPS_{X,Y}$) */
      for each $X_i$ in $p$, $1 \le i \le k$ do
         for each edge $(c,d)$ in $TDPS_{X_0,X_i}$ do
            let $(X_0.a, X_i.b) = (c,d)$ in
               for each occurrence $X_i'$ of $X_i$ in $p$ do
                  if $(X_0.a, X_i'.b) \notin TDP_p$ then AddArcTrans($TDP_p$,$(X_0.a, X_i'.b)$) fi
               od
            ni
         od
      od
      /* add  dependencies among attributes of each symbol $X$ due to the partitions of $X$ */
      for each nonterminal occurrence $X_i'$ of $X_i$ in $p$ do
         for each $X_i'.a$ do
            for each $X_i'.b$ do
               /* Kasten's partitioning algorithm places the attributes that are to */
               /* be evaluated first in the largest-numbered partition */
               if PARTITION[$X_i.a$] > PARTITION[$X_i.b$] then
                  AddArcTrans($TDP_p$,$(X_i.a, X_i.b)$)
               fi
            od
         od
      od
   od
end
```

Figure 4-11: *Algorithm to Compute TDP*

4.3.2. Computation of Relative Order Among Plans

For each two productions, $p: X_0 \rightarrow X_1 \cdots X_m$ and $q: Y_0 \rightarrow Y_1 \cdots Y_n$, such that $X_0 \xrightarrow{+} Y_0$, we want to compute:

- Index in *Plan*[p] where control is transferred after a "Visit parent" instruction in *Plan*[q].

This information is computed once for each grammar, and stored in the table *MapVisitParentToPlanIndex*. *MapVisitParentToPlanIndex*[p, q, i] returns the index of the next instruction in the plan for p to be executed after the "visit parent" instruction in position i in the plan for q.

This table is used in the algorithm for determining whether an instruction i_1 in plan p_1 associated with treenode n_1 occurs before another instruction i_2 in plan p_2 associated with treenode n_2 in the computation sequence of the semantic tree T, as follows:

(Step 1) Find the least common ancestor (*LCA*) of n_1 and n_2.

(Step 2) Find the next "visit parent" instruction following i_1 in plan p_1. Denote this instruction by vp_1.

(Step 3) Let $j = i_1$ if n_1 is equal to the *LCA*, otherwise $j = MapVisitParentToPlanIndex[LCA, n_1, vp_1]$.

(Step 4) Find the next "visit parent" instruction following i_2 in plan p_2. Denote this instruction by vp_2.

(Step 5) Let $k = i_2$ if n_2 is equal to the *LCA*, otherwise $k = MapVisitParentToPlanIndex[LCA, n_2, vp_2]$.

(Step 6) If $j < k$, then instruction i_1 comes before instruction i_2 in the computation sequence of T; otherwise i_2 comes before i_1.

This algorithm for comparing the position of two instructions in the computation sequence has amortized cost $O(\log n)$ — the cost of a least common ancestor operation [Sleator 83] — reducing the cost of the scheduling overhead incurred by the POAG evaluator for multiple asynchronous subtree replacements to $O(|ASYNC\text{-}AFFECTED| \cdot \log n \cdot k)$ for k subtree replacements.

Before describing the algorithm for constructing *MapVisitParentToPlanIndex*, we present an algorithm to compute the *ANCESTOR* relation for pairs of productions in the grammar needed for the construction of the table, where

$$ANCESTOR = \{(p,q) \mid p,q \text{ are productions, and } p \text{ is an ancestor of } q \text{ in some derivation tree of the grammar}\}.$$

The algorithm, *Ancestor*, is shown in figure 4-12.

A directed graph G is used, initially containing vertices representing the productions of the grammar and no edges. First, edges are added to G to represent the *PARENT* relation between pairs of productions — an edge between p and q indicates that one of the right hand side symbols of p derives q directly. The edges added in this step are blue. Then the transitive closure of G is computed to give the *ANCESTOR* relation. Edges added to transitively close G are red. Edge color is used in the next algorithm.

```
procedure Ancestor($G$: a directed graph);
declare
   $V$      : set of vertices of $G$;
   $E$      : set of edges of $G$;
   $p, q$   : productions;
   $X_i, Y_j$ : nonterminal symbols;
begin
   $V := \{p \mid p \text{ is a production}\}$;
   $E := \emptyset$;
   for each vertex $p: X_0 \rightarrow X_1 \cdots X_m$ in $G$ do
      for each vertex $q: Y_0 \rightarrow Y_1 \cdots Y_n$ in $G$ do
         if $X_i = Y_0,\ i = 1, \ldots, m$ then
            AddBlueEdge($p, q$) to $G$
         fi
      od
   od
   Compute transitive closure of $G$, adding red edges
end
```

Figure 4-12: *Algorithm to Compute ANCESTOR Relation*

The algorithm *BuildMaps*, shown in figure 4-13, builds the table *MapVisitParentToPlanIndex*. In order to construct *MapVisitParentToPlanIndex*, it creates another (temporary) table — *MapVisitChildToPlanIndex*. *MapVisitChildToPlanIndex*[p,q,i] returns the index of the next instruction in the plan for q to be executed after the "visit child" instruction at index i in the plan for p.

```
procedure BuildMaps();
declare
  p                          : production $X_0 \rightarrow X_1 \cdots X_m$;
  q                          : production $Y_0 \rightarrow Y_1 \cdots Y_n$;
  r                          : production $Z_0 \rightarrow Z_1 \cdots Z_k$;
  pIndex, qIndex, rIndex : integers, used as indices into plans for p, q and r respectively;
  EdgeList                   : list of edges;
begin
  EdgeList := sort edges (p,q) in ANCESTOR graph in increasing order of length of path
            of blue edges between p and q;
  for each edge (p,q) in EdgeList do
    if (p,q) is blue then
      let i be the index of the right hand side (RHS) symbol of p
            such that $X_i = Y_0$, $i = 1, \ldots, m$ in
        qIndex := 1;
        for pIndex := 1 to Length(Plan[p]) do
          if Plan[p][pIndex] = "Visit Child i" then
            MapVisitChildToPlanIndex[p,q,pIndex] := qIndex;
            while Plan[q][qIndex] ≠ "Visit parent" do qIndex := qIndex + 1 od
            MapVisitParentToPlanIndex[p,q,qIndex] := pIndex + 1;
            qIndex := qIndex + 1
          fi
        od
      ni
    else /* (p,q) is red, a transitive edge */
      let r be a production such that (p,r) and (r,q) are edges in ANCESTOR
            and (p,r) is a blue edge, and
        i be the index of the RHS symbol of p such that $X_i = Z_0$, $i = 1, \ldots, m$
            and $Z_j \overset{+}{\rightarrow} Y_0$, $j = 1, \ldots, k$, in
        for pIndex := 1 to Length(Plan[p]) do
          if Plan[p][pIndex] = "Visit Child i" then
            rIndex := MapVisitChildToPlanIndex[p,r,pIndex];
            while Plan[r][rIndex] ≠ "Visit child j" do rIndex := rIndex + 1 od
            MapVisitChildToPlanIndex[p,q,pIndex] :=
                  MapVisitChildToPlanIndex[r,q,rIndex];
                  qIndex := MapVisitChildToPlanIndex[r,q,rIndex];
            while Plan[q][qIndex] ≠ "Visit parent" do qIndex := qIndex + 1 od
            rindex := MapVisitParentToPlanIndex[r,q,qIndex];
            while Plan[r][rIndex] ≠ "Visit parent" do rIndex := rIndex + 1 od
            MapVisitParentToPlanIndex[p,q,qIndex] :=
                  MapVisitParentToPlanIndex[p,r,rIndex];
            rIndex := rIndex + 1
          fi
        od
      ni
    fi
  od
end
```

Figure 4-13: *Computation of MapVisitParentToPlanIndex*

BuildMaps first sorts the edges (p,q) in the *ANCESTOR* relation in increasing path-of-blue-edges order, that is, first the pairs of productions such that p is the parent of q are considered, (length of path-of-blue-edges is 1), then those such that p is the grandparent of q (length of path-of-blue-edges is 2), and so on. Then, the algorithm iterates over the sorted list of edges, considering them one at time.

If the edge considered, (p,q), is blue (a direct edge), then the actions of the evaluator are simulated to find the instruction i in q's plan that is executed after each "visit child" instruction in p's plan, where the child visited is the left hand side symbol of q. Let j be the first "visit parent" instruction after instruction i in q's plan. *MapVisitParentToPlanIndex*$[p,q,j]$ is then initialized to the index of the instruction following the "visit child" instruction in p's plan.

If the edge (p,q) is red (a transitive edge), then the principle of dynamic programming is used. We find a production r such that (p,r) and (r,q) are edges in *ANCESTOR*, and (p,r) is a blue edge. The length of the path-of-blue-edges of both (p,r) and (r,q) is less than that of (p,q), and therefore the table entries for these pairs of plans must be already filled in. We can use the table entries in the *MapVisitChildToPlanIndex* to determine where a "visit child" in p's plan takes us in q's plan: (1) find the next instruction in r's that will be executed following the "visit child r" instruction in p's plan (from table); (2) find the next "visit child s" instruction in r, where s is an ancestor of the left hand side symbol of q; (3) find the next instruction in q's plan that will be executed following the "visit child s" instruction in r's plan (from table). The entry in the *MapVisitParentToPlanIndex* table is then computed similarly to what we explained for direct edges.

4.4. Related Work

The class of ordered attribute grammars was defined by Kastens, who also described polynomial time algorithms for constructing evaluators for them [Kastens 80]. Yeh describes an incremental version of Kasten's evaluator [Yeh 83]. The evaluator used in the Cornell Synthesizer Generator for ordered attribute grammars is presented in [Reps 89a]. This algorithm is also based on Kasten's, and is similar to Yeh's. Both these incremental algorithms only allow single subtree replacements.

Yeh and Kastens extended the algorithm for single subtree replacements reported in [Yeh 83] to handle multiple synchronous subtree replacements [Yeh 88]. As we mentioned in the related work section of the previous chapter, such an algorithm is not effective for evaluating changes to the semantic tree that occur asynchronously because it requires the changes to be batched up and then evaluated, obviating the benefits of the incremental nature of these algorithms.

Peckham independently devised static incremental evaluators for multiple synchronous or asynchronous subtree replacements for *globally partitionable* attribute grammars [Peckham 90]. The class of globally partitionable attribute grammars is comparable to the pairwise ordered attribute grammar class defined in this chapter. His algorithm maintains a structure tree, which is a ''structure-preserving projection of a tree onto a vertex subset'', to skip over nodes in the tree not affected by any of the subtree replacements. The structure tree contains all nodes with changed attributes, as well as their least common ancestors. The structure tree is modified during evaluation as additional nodes whose attributes must be reevaluated are encountered. Peckham's algorithms minimize the number of attributes evaluated. The worst-case bookkeeping costs incurred by the synchronous and asynchronous evaluators are $O(|ASYNC\text{-}AFFECTED| \cdot \log n \cdot k)$ and $O(|ASYNC\text{-}AFFECTED| \cdot n \cdot k)$, respectively, for k subtree replacements.

Parallel incremental attribute evaluation techniques for ordered attribute grammars are described in [Zaring 90]. Two parallel evaluation algorithms are presented. In the ''synchronous'' version, a process is forked for each attribute that is ready for evaluation, *i.e.*, those attributes whose arguments have already been evaluated. In the ''asynchronous'' version, a process is forked for any arbitrary attribute evaluation, but this process may have to wait if one of its arguments is not yet available.[21] Zaring's algorithms are applicable only to single subtree replacements.

[21]Note that the terms ''synchronous'' and ''asynchronous'' are used by Zaring to describe the synchronization technique of the parallel algorithm, and not to characterize the occurrence of subtree replacements.

Chapter 5

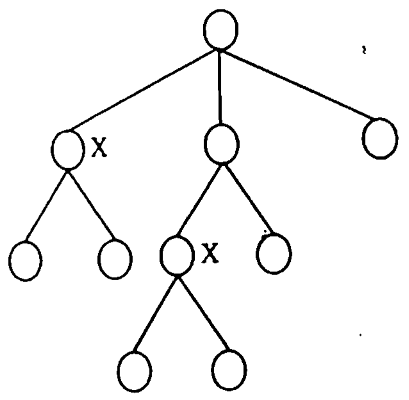
Extending Attribute Grammars to Support Static Semantic Analysis for Programming-in-the-Large

5.1. Introduction

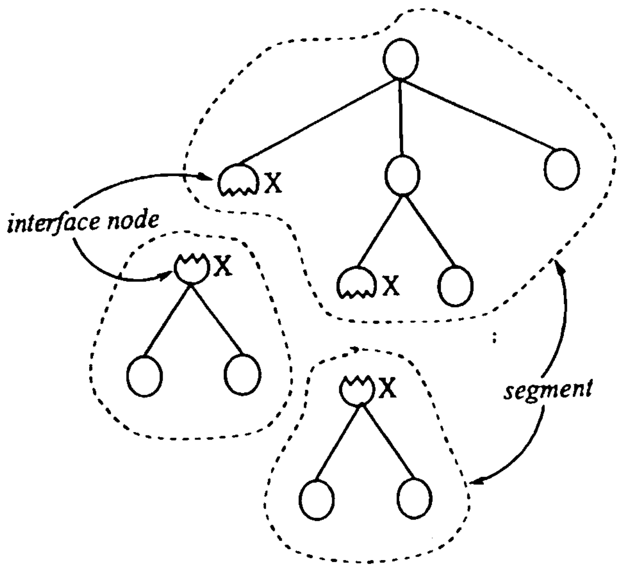
In this and the following two chapters we are concerned with programs that are composed of a number of modular units, such as Ada library and secondary units or C source and header files. Modern programming systems, such as for Ada and C, often provide specific language constructs or environmental conventions for defining separate modular units and the composition of a program in terms of these units. The abilities to formally specify the structure of a large program and analyze the interface semantics among modular units are crucial for programming-in-the-large, but have previously been addressed primarily by module interconnection languages (*e.g.*, [DeRemer 76, Tichy 79, Habermann 81, Wolf 85, Narayanaswamy 87]) independent and distinct from the formalisms used to define the programming language structure and semantics within the modular units. We address the problem of unifying the approaches to formalizing and analyzing both inter-module and intra-module static semantic properties.

The use of the attribute grammar formalism for specifying the static semantic analysis (and code generation) of monolithic programs is well-understood (*e.g.*, [Ganzinger 77, Reps 84b, Waite 84, Farrow 84, Jourdan 90]). However, attribute grammars for languages with programming-in-the-large facilities, such as for Ada [Uhl 82], define only the intra-module semantics since the formalism cannot directly express inter-module semantics. Thus normal English prose or a secondary formalism must be used to describe inter-module semantics, either preventing automatic analysis mechanisms or requiring an *ad hoc* integration of two distinct mechanisms, respectively. We fulfill our goal of unifying inter-module and intra-module static semantics by *extending* attribute grammars to express the programming-in-the-large constructs found in real programming languages, including textual inclusion, multiple kinds of compilation units, and nested compilation units.

In our extended attribute grammar formalism, a program that is composed of multiple modular units is represented by an attributed *segmented derivation tree.* A segmented derivation tree is a derivation tree that is decomposed into segments at some of its nodes, where each segment represents a single modular unit. For example, figure 5-1 (a) shows a derivation tree that is decomposed into segments at the nodes marked *X*, resulting in the three segments shown in figure 5-1 (b). A derivation tree node on the boundary of two segments is called an *interface* node, and it is replicated in both of the segments. The segment where the interface node appears as a leaf node is called the *parent segment* with respect to that interface node; the segment where the interface node is the root node is called the *child segment.* Two segments connected at an interface node are called *adjoining* segments.



(a): *A Derivation Tree*



(b): *A Segmented Derivation Tree*

Figure 5-1: *Segmented Derivation Tree*

The structure of the derivation tree representing a program is specified by the context-free grammar describing the syntax of the language in which the program is written. However, context-free grammars are not sufficiently expressive, and cannot directly specify segmented derivation tree structures. We extend CFGs to

- Denote which nonterminal symbols are interface nodes, and
- Allow the specification of the following types of segment interconnections:
    - Unordered collection of segments; *e.g.*, Ada library units, C files, Modula-2 modules.
    - Included shared segments; *e.g.*, C header files.
    - Included nonshared segments; *e.g.*, Ada subunits.

We extend the definitions of attributes and the built-in operators available for use in their semantic equations, to employ extended context-free grammars and express the static semantic properties of segment interfaces. We describe how existing attribute evaluation algorithms can be extended to *local segment evaluators* for attribute evaluation within segments, and combined with a *global evaluator* for intersegment linkage and propagation of attribute values across interface nodes. Such a segmented attribute evaluator can be used to decorate the segmented derivation tree with attributes for the purpose of detecting interface errors (and generating code). Figure 5-2 depicts a pattern of attribute flows across the segment boundaries, with the attributes of interface nodes replicated in both segments.

Our result is a uniform approach to formal specification of both intra-module and inter-module static semantic properties, that is, both within and between segments, with the ability to use attribute evaluation algorithms to carry out a complete static semantic analysis of a multi-module program.

The rest of this chapter and the following two chapters are organized as follows. In section 5.2 of this chapter, we introduce segmentable context-free grammars, illustrating with examples from Ada, Pascal and C. Algorithms for transforming the segmented representation of a program as specified by a segmentable context-free grammar into an alternative, but more convenient, representation specified by a related grammar are presented in section 5.3 of this chapter.

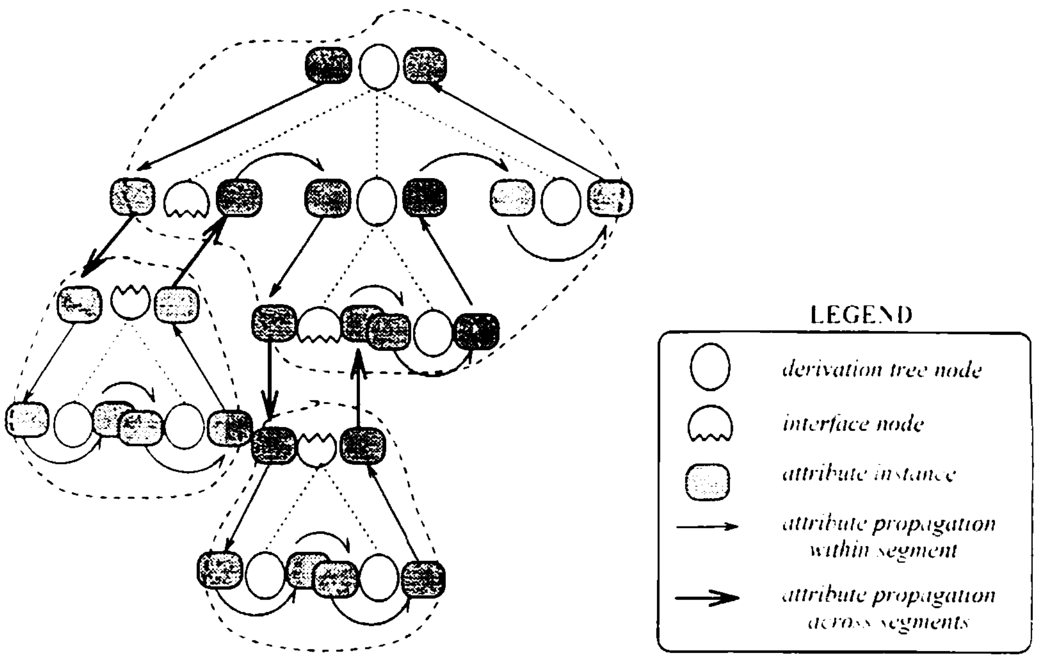

LEGEND
derivation tree node
interface node
attribute instance
attribute propagation within segment
attribute propagation across segments

Figure 5-2: *Segmented Attribute Evaluation*

Segmentable context-free grammars provide the underlying substrate for segmentable attribute grammars, which are the topic of chapter 6. Besides defining this class of attribute grammars, in this chapter we also discuss the issues of attribute evaluation in the context of programming-in-the-large, particularly the separation of concerns between the local evaluator for each segment and the global evaluator that propagates attribute flows across segment boundaries. The approach described in chapter 6 applies to the general class of noncircular attribute grammars, extended as we describe.

We focus in chapter 7 on a particular anomaly that arises when attributes flow from one segment into another and then back to the first segment, and describe a technique for *summarizing* attributes in such a way that evaluation is not delayed in the first segment due to the propagation through the second segment. Our summarizing technique involves the transformation of an attribute grammar into an equivalent one, and is only applicable to a subclass of attribute grammars that we define. We conclude in section 7.3 with a discussion of other work related to these three chapters.

5.2. Segmentable Context-Free Grammars

A *segmentable context-free grammar* is an extension of a context-free grammar, and is denoted as $G^+ = (N, T, S, D, E, P^+)$. N and T are finite sets of nonterminal and terminal symbols, respectively, and S is the start symbol of the grammar; these have the same meaning as for a context-free grammar. The other components of a segmentable CFG are defined as follows.

- D is a finite set of nonterminal property declarations. These declarations are given in the CFG in either of the following two forms:

  (i) **distributable** X

  (ii) **shared distributable** X

  where $X \in N$ and $X \neq S$. A nonterminal property declaration states that the nonterminal symbol X derives a segment; or, in other words, X is an interface node between two segments. In the first form of a nonterminal property declaration, the child segment derived from X interfaces to exactly one parent segment that has X as a leaf node. In the second form, a child segment derived from X may interface to several parent segments each of which has X as a leaf node.

  The start symbol of the grammar, S, derives the top-level segment in the segmented derivation tree representation of the program. In the typical case where a program is implicitly constructed from an unstructured collection of modules, this top-level segment may correspond to the ''makefile'' [Feldman 79] rather than a language construct.

- E is a singleton set containing the **set-of** construct. This construct may be used on the right-hand side of productions in the grammar to describe an unordered collection of segments.

  $E = \{\,\textbf{set-of}(X)\,\}$

  where $X \in N$, and X is a nonshared distributable symbol.

- P^+ is a finite set of productions, which are of the form:

  $Y \rightarrow \alpha$

  where $Y \in N$, and $\alpha \in (N \cup T \cup E)^*$.

In the next subsections, we show how segmentable context-free grammars can be written for the languages Ada, C, and Pascal, to indicate how programs in these languages may be divided into segments.

5.2.1. Example 1: Ada

In an Ada program, a segment corresponds to an Ada compilation unit as defined in the Ada reference manual [AdaTEC 82]. We identify three classes of compilation units, and define a different segment type to represent each class. The three segment types are specification segments, implementation segments, and subunit segments. A *specification segment* is either a package specification, a subprogram specification, or a subprogram body that has no corresponding specification; this segment type corresponds to the library units in the Ada reference manual. An *implementation segment* is either a package body or a subprogram body that has a corresponding specification, while a *subunit segment* is a subunit; implementation and subunit segments are collectively called secondary units in the Ada reference manual.

The extended context-free grammar in figure 5-3 defines the three kinds of segments in Ada and their interconnections. (Nonterminals shown in *italics* are defined in the Ada reference manual.) There are three productions defining a specification segment, one for each kind of compilation unit represented by this segment type. A specification segment that derives a subprogram specification must have a corresponding body; this is indicated in production *p2* by the interface nonterminal *implementation_segment*, which serves as the connection point between this specification segment and the implementation segment representing the subprogram body.

```
p1:  ada_program ::= set-of (specification_segment);

     distributable specification_segment;
p2:  specification_segment ::= context_clause subprogram_declaration implementation_segment
p3:            | context_clause package_declaration [ implementation_segment ]
p4:            | context_clause subprogram_body;

     distributable implementation_segment;
p5:  implementation_segment ::= context_clause subprogram_body
p6:            | context_clause package_body;

p7:  subunit_body_stub ::= body_stub subunit_segment;

     distributable subunit_segment;
p8:  subunit_segment ::= context_clause subunit;
```

Figure 5-3: *Extended CFG for Definition and Interconnection of Segments in Ada*

A specification segment that derives a package specification may or may not have a corresponding body; in this case, shown in production *p3*, the interface nonterminal *implementation_segment* connecting the specification segment to the implementation segment representing the package body is enclosed in square brackets ([. . .]), indicating that it is optional. A specification segment that derives a subprogram body has no corresponding implementation segment, and thus there is no need for an interface nonterminal on the right-hand side of production *p4*.

To keep the grammar in figure 5-3 short, we do not distinguish between implementation segments that represent subprogram bodies and those that represent package bodies. The result is that interfacing a package body with a subprogram specification, or *vice versa*, is syntactically correct according to this grammar. To disallow this, the semantic analysis specified by the AG could check that a package (subprogram) body is only connected to a package (subprogram) specification, and flag an error otherwise. Alternatively, the grammar of figure 5-3 could be rewritten so that there are two kinds of implementation segments, one for subprogram bodies and one for package bodies, with the appropriate symbol used in the interface constructs of productions *p2* and *p3*.

The connection between a subunit segment and its parent is defined in production *p7* in figure 5-3. Each subunit in Ada, represented as a subunit segment, must have a corresponding body stub. The body stub is a declarative item in the parent segment, which specifies that the body of a subprogram, package or task declared in the parent is to be developed as a separate unit (typically a separate file in conventional development). Production *p7* associates with each body stub an interface to the corresponding subunit segment. All occurrences of *body_stub* that appear on the right-hand side of productions in the syntax given in the Ada reference manual are replaced by the nonterminal *subunit_body_stub*.

5.2.2. Example 2: C

Figure 5-4 gives an extended context-free grammar for the C language. A C program is composed of a collection of source files, as specified in production *p1*. A source file contains a list of definitions, such as function definitions, data definitions, or include definitions. An include definition corresponds to the C include statement, such as the statement *#include "foo.h"*, which specifies that the contents of *foo.h* are included as text in the source file in which the include statement appears. The connection between the segment for the source file containing the include statement and the segment representing the included file is specified by the interface nonterminal *c_header_file* in production *p9*. This interface symbol is shared since the file *foo.h* may be included in more than one source file. Production *p10* specifies that a segment that corresponds to an included file is also composed of a list of definitions.

```
p1:  C_program ::= set-of (c_file);

     distributable c_file;
p2:  c_file ::= c_defs;

p3:  c_defs ::= /* empty */
p4:            | c_def c_defs;

p5:  c_def ::= function_def
p6:            | data_def
p7:            | data_type_def
p8:            | include_def
               | . . . ;

p9:  include_def ::= file_name c_header_file;

     shared distributable c_header_file;
p10: c_header_file ::= c_defs;
```

Figure 5-4: *Extended CFG for Definition and Interconnection of Segments in C*

5.2.3. Example 3: Pascal

The segmentable CFG for a Pascal program shown in figure 5-5 specifies that each procedure declaration declared in the outermost scope of a Pascal program is a separate segment.[22] The segments representing outer-level procedures form an ordered collection (or sequence). This facilitates writing an attribute grammar for the Pascal scope rules to check that a procedure is declared before it is called,[23] which would be a problem if procedures were modeled as an unordered collection of segments. As seen from this example, no new construct is necessary to specify an ordered collection of segments.

```
p1:  pascal_program ::=  name program_params label_decls const_decls type_defs
                              var_decls proc_decls stmts;

p2:  proc_decls ::= /* empty */
p3:             | proc proc_decls;

     distributable proc;
p4:  proc ::= ...;
```

Figure 5-5: *Extended CFG for Definition and Interconnection of Segments in Pascal*

Let us look in some more detail at the Pascal example to show how flexible an extended CFG is in describing a segmented derivation tree structure. A Pascal program in the language specified in figure 5-5 must have each outermost procedure in a separate segment, and no outermost procedure may be declared in the main program segment. These two restrictions may be removed by minor changes in the extended CFG, as shown in figure 5-6.

The grammar in figure 5-6 (a) defines outermost procedures in a Pascal program in terms of a list of lists of procedure declarations. A segment in this grammar consists of a list of procedure declarations, and thus may contain an arbitrary number of procedures. In contrast, the grammar in figure 5-6 (b) specifies that outermost procedures do not have to be developed as separate segments, but may be declared either in the main program segment or as a separate segment. This is accomplished by

[22]This requires that a different nonterminal symbol be used for nested procedures or functions, which are not shown in figure 5-5.

[23]This would be similar to the AG given in figure 2-1 in chapter 2, section 2.1, that checks that an identifier is not declared more than once.

```
p1:  pascal_program ::=  ... proc_decls_list ... ;

p2:  proc_decls_list ::= /* empty */
p3:           | proc_decls_segment proc_decls_list;

     distributable proc_decls_segment;
p4:  proc_decls_segment ::= proc_decls;

p5:  proc_decls ::= /* empty */
p6:           | proc proc_decls;

p7:  proc ::= ... ;
```

(a): *Segments with More than One Procedure*

```
p1:  pascal_program ::=  ... proc_decls ... ;

p2:  proc_decls ::= /* empty */
p3:           | proc1 proc_decls
p4:           | proc2 proc_decls;

     distributable proc1;
p5:  proc1 ::= proc;
p6:  proc2 ::= proc;

p7:  proc ::= ... ;
```

(b): *Optional Distributable Procedure Segment*

Figure 5-6: *Other Segmentation Structures for Pascal*

having two nonterminal symbols for an outermost procedure, *proc1* and *proc2*, one of which is distributable and the other is not. We call this an *optional distributable* segment.

A Pascal program that has been segmented according to one grammar (such as the one shown in figure 5-5) can be automatically transformed into a program with a different segmentation scheme (such as either of those shown in figure 5-6). This feature is important for a programming environment since program development often entails changing the modular structure of the program. The grammar writer would only need to specify the simple segmentation scheme shown in figure 5-5, and the associated semantic equations, and the extended grammar would be automatically transformed to permit a more flexible organization of the program. The transformation algorithms are described in the following section.

5.3. Transforming the Segment Organization of a Program

In this section we describe two kinds of transformations of the segment organization of a program specified by a segmentable context-free grammar. The first, called *list segment transformation*, takes a program containing a sequence of segments, where each segment's root nonterminal is X, and transforms it to a program containing a sequence of segments, where each segment contains a list of subtrees each with root nonterminal X. The second transformation, called *optional segment transformation*, takes a program containing a sequence of segments with root X, and transforms it to a program containing a sequence of nodes from which are derived segments or subtrees rooted at X. These two transformations are illustrated in figure 5-7. These transformations entail changes to both the segmentable CFG as well as the semantic equations associated with the changed productions.

5.3.1. List Segment Transformation

Let $G_1^\dagger = (N_1, T, S, D_1, E, P_1^\dagger)$ denote a segmentable CFG deriving programs containing a sequence of segments with root X, and *X-List* the nonterminal symbol in N_1 deriving the list of segments. (The nonterminal X corresponds to *proc* in the Pascal grammar of figure 5-5 while *X-List* corresponds to *proc_decls*.) The relevant productions in $G_1^\dagger$ have the following form:[24]

```
p:   ... ::= ... X-List ...;

q:   X-List ::= /* empty */
r:            | X  X-List;

     distributable X;
s:   X ::= ... ;
```

Let $G_2^\dagger = (N_2, T, S, D_2, E, P_2^\dagger)$ be the segmentable CFG resulting from the list

[24]For simplicity, any additional symbols on the right-hand side of productions q and r are omitted. The transformation algorithms can easily be modified to take into account these extra symbols.

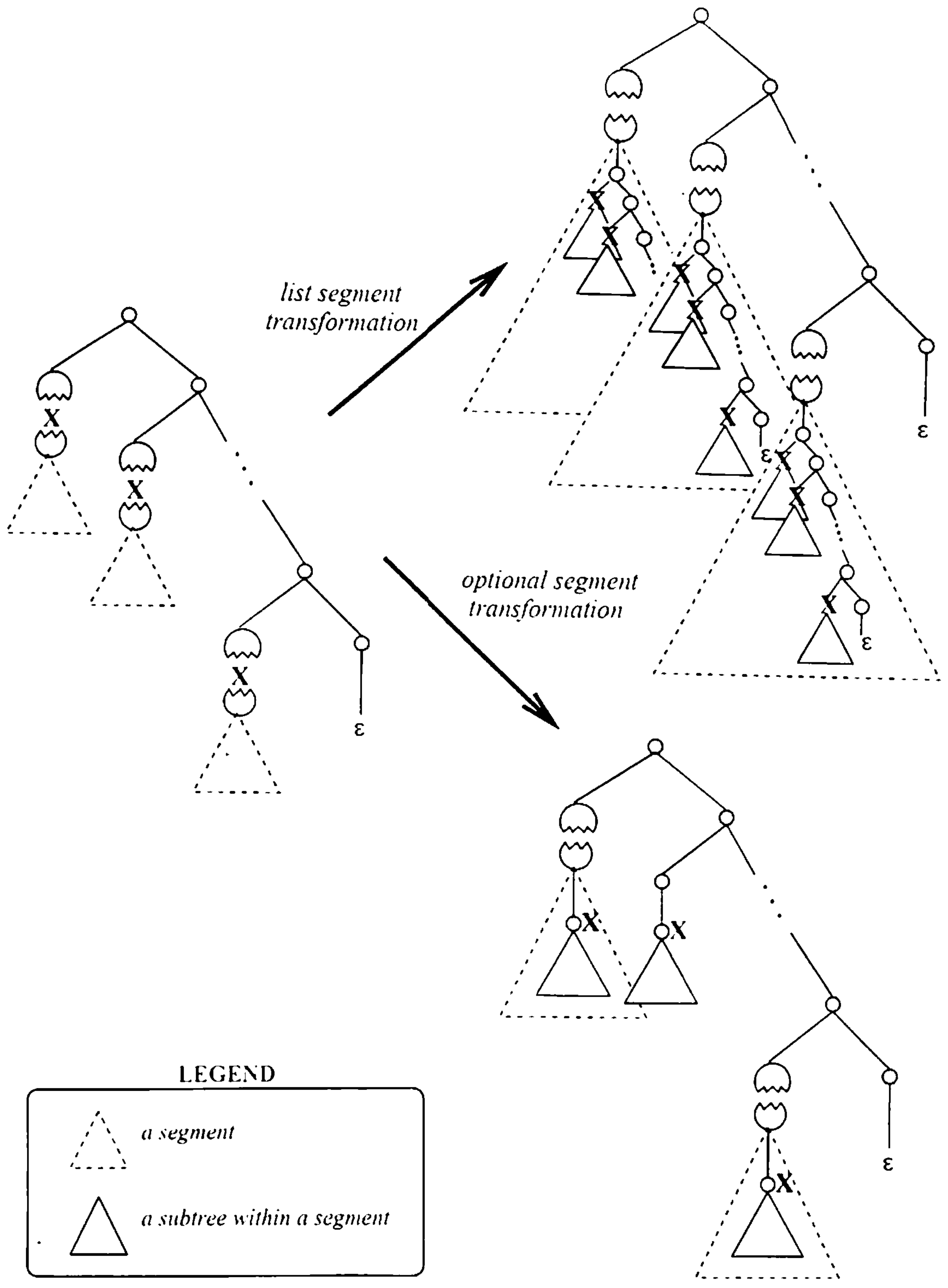

list segment transformation
optional segment transformation
X
ε
LEGEND
a segment
a subtree within a segment

Figure 5-7: *Transformation of Segment Organization of a Program*

transformation. That is, $G_2^{\ddagger}$ derives programs containing a sequence of segments, each of which contains a list of subtrees with root X.

Algorithm to transform $G_1^{\dagger}$ to $G_2^{\ddagger}$:

1. Initialize $G_2^{\ddagger}$ to $G_1^{\dagger}$.
2. Add new nonterminal symbols *X-List-List* and *X-List-Segment* to N_2.
3. Let p denote any production in $P_1^{\dagger}$ where *X-List* appears on the right-hand side but not on the left-hand side. Replace the occurrence of *X-List* in each production p in $P_2^{\ddagger}$ by *X-List-List.*
4. Define *X-List-List* to be a (possibly empty) list of *X-List-Segment* by adding the following two productions to $P_2^{\ddagger}$:

```
t:   X-List-List ::= /* empty */
u:               | X-List-Segment X-List-List;
```

5. Declare the nonterminal *X-List-Segment* **distributable**, instead of X, in D_2.
6. Add the following production to $P_2^{\ddagger}$:

```
v:   X-List-Segment ::= X-List;
```

The productions in $G_2^{\ddagger}$ that are changed by the list transformation are the following:

```
p:   . . . ::= . . . X-List-List . . . ;

t:   X-List-List ::= /* empty */
u:             | X-List_segment X-List-List;

     distributable X-List-Segment;
v:   X-List-Segment ::= X-List;

q:   X-List ::= /* empty */
r:          | X  X-List;

s:   X ::= . . . ;
```

One may wonder why it is necessary to introduce the new symbol *X-List-Segment* and the unit production v. That is, why not define *X-List-List* to be a list of *X-List*, and declare *X-List* the distributable symbol? The reason is that the node labeled with the symbol *X-List* in each instance of production r in the derivation tree would become the root of a new segment. Each of these *X-List* segments would contain exactly one subtree rooted at X, which is not what is intended by the list transformation.

The above transformation of the segmentable CFG must be accompanied by a complimentary transformation of the semantic equations associated with the relevant productions in the grammar, which is given in the algorithm below. Recall that $A(X)$ denotes the attributes associated with the nonterminal X, $I(X)$ the inherited attributes of X, and $S(X)$ the synthesized attributes of X.

Algorithm to transform AG_1, based on $G_1^\dagger$, to AG_2, based on $G_2^\ddagger$:

1. Declare the attributes for the new nonterminals *X-List-List* and *X-List-Segment* to be the same as for *X-List*; *i.e.*, $I(X\text{–}List\text{–}List) = I(X\text{–}List\text{–}Segment) = I(X\text{–}List)$ and $S(X\text{–}List\text{–}List) = S(X\text{–}List\text{–}Segment) = S(X\text{–}List)$.

2. In the semantic equations associated with each production p in $G_2^\ddagger$, replace each attribute occurrence $X\text{–}List.a$ by $X\text{–}List\text{–}List.a$, where $a \in A(X\text{–}List)$.

3. For the two productions t and u defined above, copy the semantic equations associated with the two productions defining *X-List* (*i.e.*, productions q and r), replacing *X-List* by *X-List-List* and X by *X-List-Segment.*

4. Associate the following pair of semantic equations with the unit production v for each inherited attribute a_i in $I(X\text{–}List)$ and each synthesized attribute a_s in $S(X\text{–}List)$:

$$\text{X-List}.a_i = \text{X-List-Segment}.a_i$$
$$\text{X-List-Segment}.a_s = \text{X-List}.a_s$$

5.3.2. Optional Segment Transformation

Let $G_1^\dagger$ be a segmentable CFG defined as in subsection 5.3.1. Let $G_3^\ddagger = (N_3, T, S, D_3, E, P_3^\ddagger)$ be the segmentable CFG resulting from the optional segment transformation. That is, $G_3^\ddagger$ derives programs containing a sequence of subtrees rooted at X, where an X subtree may either be in a separate segment or may form part of the parent segment.

Algorithm to transform $G_1^\dagger$ to $G_3^\ddagger$:

1. Initialize $G_3^\ddagger$ to $G_1^\dagger$.

2. Add new nonterminal symbols $X1$ and $X2$ to N_3.

3. Let r be the recursive production in $P_1^\dagger$ defining the list of segments X. Replace production r in $P_3^\ddagger$ by the following two productions:

```
w:  X-List ::= X1 X-List
x:           | X2 X-List;
```

4. Declare the nonterminal *X1* **distributable**, instead of X, in D_3.

5. Add the following unit productions to $P_3^\ddagger$:

```
y:  X1 ::= X;
z:  X2 ::= X;
```

The productions in $G_3^\ddagger$ affected by the optional segment transformation are the following:

```
p:   . . . ::= . . . X-List . . . ;

q:   X-List ::= /* empty */
w:            | X1  X-List
x:            | X2  X-List;

     distributable X1;
y:   X1 ::= X;

z:   X2 ::= X;

s:   X ::= . . . ;
```

We now present the algorithm to transform the semantic equations associated with the affected productions in the grammar $G_3^\ddagger$.

Algorithm to transform AG_1, based on $G_1^\dagger$, to AG_3, based on $G_3^\ddagger$:

1. Declare the attributes for the new nonterminals *X1* and *X2* to be the same as for X; *i.e.*, $I(X1) = I(X2) = I(X)$ and $S(X1) = S(X2) = S(X)$.

2. For the two productions w and x defined above, copy the semantic equations associated with the recursive production r defining *X-list*, with each attribute occurrence $X.a$ replaced by $X1.a$ in production w and by $X2.a$ in production x, where $a \in A(X)$.

3. Associate the following pair of semantic equations with the unit production y for each inherited attribute a_i in $I(X)$ and each synthesized attribute a_s in $S(X)$:

$$X.a_i = X1.a_i$$
$$X1.a_s = X.a_s$$

4. Repeat step (3) for production *z*, substituting *X2* for *X1*.

The details of how these transformations are integrated into a multi-user programming environment are beyond the scope of this thesis.

Chapter 6
Segmentable Attribute Grammars

We define the class of *segmentable attribute grammars* (SAGs) to contain extended attribute grammars that are based on segmentable context-free grammars. The purpose of this chapter is twofold. First, we describe how attribution of a segmented derivation tree, that is, giving values to the attributes decorating the nodes of the derivation tree, is performed through the combined actions of local and global evaluators. This differs substantially from the attribution of a monolithic derivation tree, where only one kind of evaluator is required. Second, we present additional built-in operators that are required to write semantic equations associated with productions containing distributable symbols, or the **set-of** construct described in the previous chapter.

More formally, an SAG consists of the following components:

1. An underlying segmentable context-free grammar, G^+, which describes the structure of the segmented derivation tree of sentences in the language $L(G^+)$. (Segmentable CFGs were defined in chapter 5, section 5.2.)

2. Attribute declarations, specifying the set of attributes that are associated with each grammar symbol, and their types. There are two new concepts:
   - All attributes associated with an interface (distributable) nonterminal symbol are implicitly declared to be *interface attributes*. The value of an interface attribute is accessible in the segment in which it is defined, as well as in the adjoining

segment.[25]

- A new type of attribute, called a *conglomerate attribute*, is defined for nonterminal symbols deriving segments by the **set-of** construct. Conglomerate attributes collect information from the collection of segments connected by the **set-of** construct, and are explained in section 6.4.

3. Segment linkage declarations, which are required to specify which segment instances with interface nodes labeled with the same nonterminal symbol are connected to each other. Segment linkage will be described in section 6.2.

4. Semantic equations associated with productions in G^+. These differ from their counterparts in conventional attribute grammars in two ways:

   - New built-in operators may be used in semantic equations associated with productions containing distributable symbols.

     - The operators **makeNull**, **assign**, and **compute** are used to manipulate conglomerate attributes, and the boolean operator **isUnique** is used to check that segments in an unordered collection have unique names; these operators are defined in section 6.4.

     - The operators **linkedSetSize** and **for each linked <nonterminal>** are used to determine the number of child segments linked to a parent segment, and to iterate over the number of child segments, respectively; these operators are related to segment linkage, and are defined in section 6.2.

   - *Completing semantic equations* are required for interface attributes. Completing semantic equations give values to interface attributes defined in segments that have not yet been created, and are explained in section 6.1 below.

[25]There may exist attribute grammar applications where it is desirable that some attributes associated with an interface nonterminal symbol be local to one segment, that is, their value is accessible only in the segment in which the attribute is defined. Since this can be achieved by means of *local* attributes [Jourdan 89, Reps 89b] — attributes associated with a production rather than with nonterminal symbols — we decided to keep the definition of SAGs simple by having only interface attributes associated with an interface nonterminal symbol. If a non-interface attribute of the interface symbol X is needed in the segment where X is a leaf node, a local attribute associated with the production where X appears on the right-hand side is declared. And instead of a non-interface attribute of the interface symbol X in the segment derived from X, one can declare a local attribute in the production where X is the left-hand side symbol.

As we mentioned in earlier chapters, there is a hierarchical classification of standard attribute grammars based on the complexity of the expressible attribute dependencies, leading to different evaluation strategies for the different classes. The segmentable classification is orthogonal to the evaluation strategy classification. Thus, a segmentable attribute grammar may be ordered, in which case it can be evaluated by the algorithms described in chapter 4, modified to handle segmented derivation trees as we describe in section 6.1. Or, a segmentable AG may not be ordered but is noncircular, in which case the dynamic evaluation algorithms described in chapter 3 (also modified for segments) can be used to perform attribute evaluation.

Figure 6-1 gives an example of a segmentable attribute grammar specification of a simple modular language. A program in this language consists of a set of modules. The exported facilities of a module are stored in the attribute *exports* associated with each module. The facilities exported by all the modules are collected in the conglomerate attribute, *allexports*, associated with the entire program. This is accomplished by means of the **assign** operation. The **isUnique** operator is used to check whether a module's name is unique in the set of modules comprising the program.

A module references facilities exported by other modules through the import statement. An import statement names the module from which the facility is imported, and the facility itself. The import statement uses the component of the *allexports* attribute identified by the imported module's name to check the legality of the import statement (the imported module must exist and must be unique, and the imported facility must be exported). The **compute** operation finds the appropriate component.

(Segment linkage declarations are not required in this simple example and will be illustrated in section 6.2 where this concept is explained.)

Program:	{ synthesized attributes:	**conglomerate** allexports; }

Module:	{ synthesized attributes:
	inherited attributes:
		name, exports;
		error **default** ""; }

Name:	{ synthesized attributes:	id; }
Import:	{ synthesized attributes:	error; }
VarId:	{ synthesized attributes:	name; }

(a): *Attribute Declarations*

```
p1:  Program ::= set-of (Module);
         { for each linked Module$i
               assign(Program.allexports, Module$i.name, Module$i.exports);
           for each linked Module$i
               Module$i.error = isUnique(Module$i.name)
                       ? ""
                       : "<-- duplicate module"; }

     distributable Module;
p2:  Module ::= Name Export Import Decl Body;
         { Module.name = Name.id;
           Module.exports = ... ;
           ...; }

p3:  Import ::= ModuleId VarId
         { local single_module_exports;

           single_module_exports = compute({Program.allexports}, ModuleId.name);
           Import.error = (single_module_exports == bottom)
               ? "<-- imported module unknown"
               : (single_module_exports == multiple)
                       ? "<-- imported module duplicate"
                       : Member(single_module_exports, VarId.name)
                               ? ""
                               : "<-- variable not exported";
           ...; }
```

(b): *Abstract Syntax and Semantic Equations*

Figure 6-1: *Specification of a Simple Modular Language*

6.1. Attribute Evaluation for Segmented Derivation Trees

An attribute evaluator for a segmentable AG consists of two parts: (1) a *local segment evaluator*, which evaluates attributes within a segment, and (2) a *global evaluator*, which propagates attributes among segments and performs evaluations involving more than one segment.

A local segment evaluator is similar to an evaluator for a monolithic semantic tree except for its actions at attributes associated with the segment's interface nodes. Recall that an interface node is a node that is on the boundary between two segments, and therefore is either the root or a leaf node of a segment. Interface nodes and their associated (interface) attributes are duplicated in the semantic tree of the two adjoining segments.

The *output attributes of a segment S* are those attributes associated with interface nodes of *S* that are defined by semantic equations associated with production instances in *S*. These are the synthesized attributes of the root node of *S* and the inherited attributes of leaf interface nodes of *S*. The *input attributes of a segment S* are the copies of the output attributes of segments connected to *S*; that is, the inherited attributes of the root node of *S* and the synthesized attributes of leaf interface nodes of the segment *S*.

A local evaluation process is initiated within a segment either because of an edit performed on the segment, or because of a change made to a remote segment that has propagated (via the global evaluator) to this one. If in the process of attribute evaluation within a segment an output attribute's value changes, the local evaluator communicates the new value to the global evaluator. The global evaluator then transmits the new value to the local evaluator of the adjoining segment, where the changed attribute is an input attribute of the segment.

When an input attribute of a segment receives a new value due to propagation by the global evaluator, the segment's local evaluator initiates its own attribute evaluation process at the corresponding interface node. Changes to the interface attributes of multiple other segments may be propagated to a segment asynchronously, so another may arrive before an evaluation triggered by a previous change has completed.

Therefore, the local segment evaluator should support some kind of merging of the evaluation processes to avoid unnecessary multiple evaluations of the same attributes as described in chapters 3 and 4.

The input attributes of a segment *S* may be used as arguments in semantic equations associated with production instances in *S*. In order for *S*'s local evaluator to be able to evaluate these semantic equations, each input attribute of *S* must have a value, even when the adjoining segment where the corresponding output attribute is defined has not yet been created. This is the purpose of *completing semantic equations*: they define the default values of a segment's input attributes.

In the SAG of figure 6-1, the *error* attribute associated with the symbol *Module* is an input attribute of the segment derived from *Module*. The completing semantic equation is defined as part of the attribute declaration, in this case defining *error* to be the empty string:

```
Module:  { synthesized attributes:      name, exports;
           inherited attributes:        error default ""; }
```

Completing semantic equations are not required to define the default values of synthesized attributes associated with a distributable symbol *X* in an unordered collection, **set-of**(*X*). In the segment where these attributes are input attributes — the segment where the symbol *X* appears as a leaf node — they can only be used to construct conglomerate attributes. As shall be explained in section 6.4, a conglomerate attribute is defined as the union of components from all existing segments in the collection. The conglomerate is defined even when there are no segments in the collection. Thus, in the attribute grammar of figure 6-1, there is no need for completing semantic equations for the synthesized attributes *name* and *exports* of the nonterminal symbol *Module*.

Before any attributes can be propagated among segments, the global evaluator must determine which segments are connected to each other. This process is described in the following section. Then, in section 6.3, we discuss attribute evaluation of segments derived from shared distributable nonterminals. This is different from the attribute evaluation procedure just described because a shared segment may be linked to more than one parent segment.

6.2. Segment Linkage

One of the functions of the global evaluator is to perform *segment linkage*, that is, determine which segment instances with interface nodes labeled with the same nonterminal symbol are connected to each other. The segmentable CFG describes which segment types can be connected together, and where the root of the child segment is (logically) connected to the parent segment. But it does not describe which two segment instances are involved in a particular connection. This additional information is required in order to propagate a changed interface attribute in a parent segment to the correct child segment, and *vice versa*.

Segment linkage is specified in the extended attribute grammar by means of *segment linkage declarations*. A segment linkage declaration has the following form:

$X \{ a_i \longleftrightarrow a_s \};$

where X is a distributable nonterminal symbol; a_i is an inherited attribute of X; and a_s is a synthesized attribute of X. The types of the two attributes a_i and a_s in a segment linkage declaration must be the same. Two segments with an interface node labeled by the nonterminal symbol X are connected to each other if the values of the attributes a_i and a_s of X are the same. There are constraints on the semantic equations defining the segment linkage attributes a_i and a_s. These constraints are described in subsection 6.2.1.

We show how segment linkage is specified for an Ada program. Figure 6-2 extends the segmentable context-free grammar for Ada (shown in figure 5-3 in chapter 5) with segment linkage declarations and semantic equations defining the segment linkage attributes for implementation and subunit segments. The attributes used for linking implementation segments to their corresponding specification segments are *specification_segment_name* and *implementation_segment_name*. These attributes are assigned the simple name of the package or subprogram whose specification or body appears in the segment. We assume that the package or subprogram name is available in the attribute *name*, but omit the semantic equations defining it.

Segment linkage for subunit segments is similar, except that fully expanded names are

```
/* segment linkage declarations */

implementation_segment
     { specification_segment_name <—> implementation_segment_name };
subunit_segment
     { body_stub_name <—> subunit_name }

/* segment linkage attribute definitions */

p2:  specification_segment ::= context_clause subprogram_declaration implementation_segment
                    { implementation_segment.specification_segment_name =
                              subprogram_declaration.name; }
p3:           | context_clause package_declaration [ implementation_segment ]
                    { implementation_segment.specification_segment_name =
                              package_declaration.name; }

     distributable implementation_segment;
p5:  implementation_segment ::= context_clause subprogram_body
                    { implementation_segment.implementation_segment_name =
                              subprogram_body.name; }
p6:           | context_clause package_body;
                    { implementation_segment.implementation_segment_name =
                              package_body.name; }

p7:  subunit_body_stub ::= body_stub subunit_segment;
                    { subunit_segment.body_stub_name =
                              Concatenate(body_stub.parent_name,body_stub.simple_name);}

     distributable subunit_segment;
p8:  subunit_segment ::= context_clause subunit;
                    { subunit_segment.subunit_name =
                              Concatenate(subunit.parent_name, subunit.simple_name); }
```

Figure 6-2: *Attribute Grammar for Matching Segments in Ada*

required by the Ada language definition. The segment linkage attributes for subunit segments are *body_stub_name* and *subunit_name*. In Ada, each subunit specifies the full name of its parent unit, starting with the simple name of the ancestor library unit. The *subunit_name* attribute of a subunit segment is assigned the full parent name specified in the subunit (attribute *parent_name*) concatenated with the simple name of the subunit (attribute *simple_name*). (The semantic equations defining the attributes *parent_name* and *simple_name* are omitted in figure 6-2.) The attribute *body_stub_name* is similarly defined; in this case, the attribute *parent_name* is the fully expanded name of the segment in which the body stub appears, and *simple_name* is the name of the declarative item specified in the body stub.

In the case of Ada, it is possible to specify segment linkage using information that is available in an Ada program written according to the syntax defined in the Ada manual. This is not always possible in other languages. For example, include files in C cannot be linked to the source files that include them without requiring additional syntax. For C include files, linkage is performed on the basis of the included file's name. The file name is specified in the parent segment that contains the corresponding include statement (see production *p9* in figure 5-4, chapter 5), and so the file name can be used to initialize a_i. However, the file name does not appear in the segment corresponding to the included file (production *p10* in figure 5-4). In order to define the attribute a_s, production *p10* must be modified to include the name of the file, as follows:

p10′: c_header_file ::= file_name c_defs;

Segment linkage declarations can be omitted when there is no ambiguity about which segment instances are connected to each other. In the example grammar shown in figure 6-1, segment linkage declarations are not required since all segments derived from the symbol *Module* are connected to the top-level segment derived from *Program*. In general, segment linkage declarations are not needed for a distributable symbol X if (i) there is only one production, say production p, in the segmentable CFG with the symbol X on the right-hand side, and (ii) the left-hand side symbol of p is not derived (directly or indirectly) from a recursively defined nonterminal symbol (*i.e.*, all derivation trees of the grammar contain only one instance of production p).

6.2.1. Constraints on Segment Linkage Attributes

The semantic equations defining the segment linkage attributes a_i and a_s are constrained so that a_i only depends on attributes defined in the same segment where a_i is defined, and a_s only depends on attributes defined in the same segment where a_s is defined. The reason for this restriction is obvious — no propagation of attributes between two segments connected at an interface node X can take place until segment linkage has occurred. To state the constraints more precisely, let p and q denote the production instances in the parent and child segments respectively that apply at the interface node X, defined as follows:

$p\colon \ldots \rightarrow \ldots X \ldots$

$q\colon X \rightarrow X_1 \ldots X_n$

The attribute a_i (defined by a semantic equation associated with p) cannot depend, directly or indirectly, on synthesized attributes of the interface nonterminal X, and a_s (defined by a semantic equation associated with q) cannot depend, directly or indirectly, on inherited attributes of X.

For an arbitrary noncircular attribute grammar G, checking whether the constraints on the segment linkage attributes, a_i and a_s, of a distributable symbol X hold, requires the computation of all possible characteristic graphs of X. (Recall from chapter 2 that the characteristic graph of a symbol X of an arbitrary noncircular grammar G may be different for different derivation trees of G.) Then, the constraint on dependencies involving a_i is satisfied if there is no edge from a synthesized attribute of X to a_i in any superior characteristic graph of X, while the constraint on dependencies involving a_s is satisfied if there is no edge from an inherited attribute of X to a_s in any subordinate characteristic graph of X.

Unfortunately, computing all possible characteristic graphs of a distributable symbol X may require exponential time. We prove this complexity result by contradiction. Suppose that there exists a polynomial time algorithm for computing all the characteristic graphs of a <u>single</u> symbol of an arbitrary noncircular grammar G. Then, we can construct a polynomial algorithm for computing all possible characteristic graphs of <u>all</u> symbols in G: execute the algorithm for finding all characteristic graphs of one symbol on every nonterminal symbol in G. This implies that the number of possible characteristic graphs of all symbols in G is bounded by a polynomial. If this were true, then we can determine whether an attribute grammar G is noncircular in polynomial time by checking if every characteristic graph of every symbol in G is acyclic. However, the circularity problem of AGs is a known NP-complete problem [Jazayeri 75], so we have our contradiction.

We therefore use an approximation that can be computed in polynomial time instead of the set of possible characteristic graphs of a distributable symbol X to check the constraints on the segment linkage attributes of X. This approximation is the relation

TDS_X, which was defined in chapter 4, section 4.3. As we said in that chapter, TDS_X contains any essential dependency among the attributes of a symbol X that could be present in any derivation tree. This relation is pessimistic because all these dependencies are assumed to be present simultaneously. Because this is a pessimistic approximation, TDS_X could contain an edge representing a dependency from a synthesized attribute of the interface symbol X to a_i (or from an inherited attribute of X to a_s) that could not be present in any semantic tree derived from the grammar. This is illustrated by the attribute grammar shown in figure 6-3, which is adapted from one given in [Waite 84].

```
p1:  Z ::= X
       { X.a = 1; }

p2:  X ::= Y
       { X.b = Y.f;
         Y.c = X.a;
         Y.d = Y.e; }

p3:  Y ::= u
       { Y.e = 2;
         Y.f = Y.d; }

p4:  Y ::= v
       { Y.e = Y.c;
         Y.f = 3; }
```

Figure 6-3: *A Noncircular Attribute Grammar*

This grammar derives two trees, shown in figure 6-4. The relation TDS_Y has the dependency edges $\{(d,f), (e,d), (c,e), (c,f)\}$. Note that both edges (d,f) and (c,e) are included in TDS_Y even though it is clear from figure 6-4 that only one of them can occur in any derivation tree. The result is that the relation TDS_X contains the edge (a,b) even though there is no dependency from $X.a$ to $X.b$ in any derivation tree of the grammar. We believe that the definitions of the segment linkage attributes a_i and a_s are simple and do not involve such pathological chains of transitive dependencies so that the approximation will work for most practical cases. See, for example, the segment linkage declarations for Ada presented above.

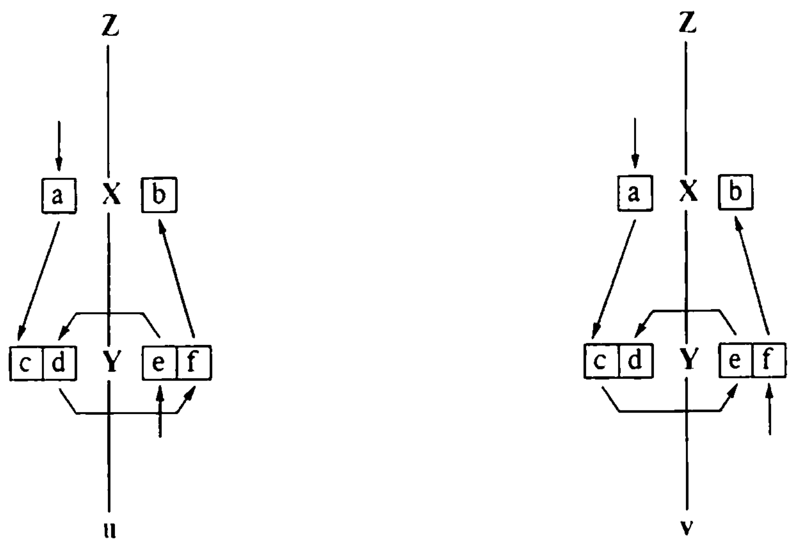


Figure 6-4: *Dependency Graphs of Trees Derived from AG of Figure 6-3*

6.2.2. Built-in Operators Related to Segment Linkage

There are two built-in operators related to segment linkage that may be used in semantic equations of a segmentable AG: **linkedSetSize** and **for each linked <nonterminal>**.

1. **linkedSetSize**(*X*): Returns the number of segments with root node labeled *X* that are linked to a parent segment at a leaf interface node labeled *X*. The argument *X* must be a distributable nonterminal symbol.

   The meaning of the **linkedSetSize** operator depends on the form of the associated production instance.

   - If the associated production is *p*: ... $\rightarrow$ ... **set-of**(*X*) ... , **linkedSetSize**(*X*) returns the number of segments *X* in the unordered collection linked to the parent segment containing production instance *p*.

   - If the associated production is *q*: ... $\rightarrow$ ... *X* ... , then **linkedSetSize**(*X*) returns the number of segments derived from *X* that are linked to the parent segment at the interface node labeled *X* in the production instance *q*. In the latter case, a return value greater than 1 indicates an error, as we shall see in the examples below.

2. **for each linked** *X*$i: Iterates over all segments derived from *X* in a specified collection of segments. Again, the collection of segments

depends on the form of the associated production instance. If the associated production is *p* defined as in (1) above, then this operator iterates over all segments in the unordered collection specified by the **set-of** construct. If the associated production is *q* defined as in (1) above, the iteration is over all child segments that are linked to the parent segment at interface node *X*.

The notation "X$i" is used in this operator to underscore the fact that there are multiple occurrences of the interface symbol *X*, and that the semantic equation in the body of the iterator is applied to each occurrence of *X*.

Examples of the use of the iteration operator are seen in the simple SAG of figure 6-1. In the first semantic equation associated with production *p1*, the conglomerate attribute *allexports* is constructed from the exported facilities of each module in the collection of modules comprising the program by the repeated application of the **assign** operator on each module. The second semantic equation associated with the same production also iterates over all modules in the program, in this case to check whether each module's name is unique.

To exemplify the use of the **linkedSetSize** operator, we show semantic equations that check that there is exactly one subunit body corresponding to each declared body stub in an Ada program.[26] The second semantic equation associated with production *p7* in figure 6-5 checks that there is at most one subunit segment for each declared body stub. This equation would flag an error if an Ada program contains two subunits defining the body of a subprogram *P* whose body stub is specified in a parent compilation unit *TOP*. The third semantic equation shown in figure 6-5 checks that there is at least one subunit segment for each declared body stub. If an Ada program contains a body stub for a subprogram *P* in the compilation unit *TOP*, but there is no subunit defining the body of *P*, this semantic equation would cause an error to be reported.

[26]In [Micallef 90], we present a SAG specification for performing inter-module semantic analysis for Ada, including (1) determining which contexts are accessible to a compilation unit and propagating context information accordingly, (2) detecting compilation units with duplicate names where unique names are required, and detecting missing bodies when they are required by specifications, and (3) checking that there is an order for submitting the compilation units comprising a program for compilation.

```
p7:  subunit_body_stub ::= body_stub subunit_segment;
                  { local no_linked_segments;

                    no_linked_segments = linkedSetSize (subunit_segment);
                    for each linked subunit_segment$i
                           subunit_segment$i.error =
                                  (no_linked_segments == 1)
                                         ? ""
                                         : (no_linked_segments > 1)
                                                ? "<-- duplicate subunit";
                    subunit_body_stub.error = (no_linked_segments == 0)
                                         ? "<-- missing subunit for this body stub"
                                         : ""; }
```

Figure 6-5: *Checking for Duplicate or Missing Subunits in Ada*

6.3. Representation and Attribute Evaluation of Shared Segments

The **shared distributable** form of a nonterminal property declaration presents additional complexity to the local attribute evaluation algorithm within a segment because the attributed derivation tree representing such a segment must contain additional information, as will be described in this section.

A segment *S* whose root node is labeled with the nonterminal symbol *X*, where *X* is declared to be a shared interface symbol, may be linked to several other segments. We call such a segment *S* a *shared segment*, and each segment linked to *S* a *shared-inclusion site of S*. For example, consider the C program in figure 6-6. This program consists of five segments: two source files (*file1.c* and *file2.c*), and three include files (*x.h*, *y.h*, and *foo.h*). The include file *foo.h* will be linked to the two source files that contain the statement *#include "foo.h"*, *file1.c* and *file2.c*.

The input attributes for such a segment *S* may have different values for each segment that includes it. Similarly, attributes in *S* that depend (directly or transitively) on the input attributes may be different. In our example, one possible input attribute of the segment *foo.h* is the list of user-defined types that have been defined prior to this point in the program, *user_defined_types_in*. This attribute has a different value depending on whether segment *foo.h* is linked to *file1.c* or *file2.c*. In the first case, *user_defined_types_in* defines the type NAME to be a character string; in the second

***** file1.c *****

```
#include <stdio.h>
#include "x.h"
#include "foo.h"

main() {
  EMPLOYEE empl1;

  printf("hello world from main\n");
  empl1.name = "Joe Blow";
  printf(empl1.name);
  f();
}
```

***** file2.c *****

```
#include "y.h"
#include "foo.h"

f(){
  EMPLOYEE worker;

  printf("hello world from f\n");
  worker.name = "Joe Blow";
  printf(worker.name);
}
```

***** x.h *****

```
typedef char *NAME;
```

***** y.h *****

```
typedef struct name {
  char *first;
  char middle;
  char *last;
} NAME;
```

***** foo.h *****

```
typedef struct employee {
  NAME name;
  int id;
} EMPLOYEE;
```

Figure 6-6: *Example Showing Why Replication is Needed for Shared Segments*

case, *user_defined_types_in* defines the type NAME to be a structure. A typical output attribute of *foo.h* is the list of user-defined types that have been defined up to and including this point in the program. Since this output attribute depends on *user_defined_types_in*, its value will also be different depending on whether segment *foo.h* is linked to *file1.c* or *file2.c*: the name field of the EMPLOYEE type is a character string in one case and a structure in the other.

Therefore, some form of replication is required for shared segments in order to represent a different collection of attribute values for each use of the shared segment. One possibility is to replicate the entire semantic tree representing the segment *S*, that is, both the derivation tree and the attributes decorating the nodes of the derivation tree. The disadvantage with this approach is that the derivation tree for *S* must be kept consistent in all the copies when changes to the segment *S* are made. A better approach is to replicate only the attributes decorating the derivation tree of *S* for each shared-inclusion site. This can be further optimized so that only attributes that may have different values for each shared-inclusion site of *S* are replicated. We describe the latter approach below.

We present algorithms to replicate and evaluate the necessary attributes of a shared segment for the following four cases: (1) creation of a new segment *S* derived from a **shared distributable** symbol; (2) replacement of a subtree within segment *S*; (3) addition of a new shared-inclusion site for segment *S* in some other segment; (4) deletion of an existing shared-inclusion site for segment *S* from some other segment. For simplicity, we assume that the AG is in normal form, and that the attribute evaluation method used is the dynamic one described in chapter 3 that applies to the general class of noncircular grammars. Static evaluation algorithms, such as the ones described in chapter 4, can be modified to handle replicated attributes of shared segments in a similar way.

There are two other cases that merit consideration, but they can be handled by one of the algorithms for the four cases above, or by completing semantic equations. (5) Deleting a shared segment; deleting a shared segment that is linked to other segment(s) requires that the completing productions be used in the other segments to give values to their interface input attributes, but does not require any special handling with regards to replication of attributes. (6) Replacing an entire shared segment; this is equivalent to deleting the shared segment (case 5) followed by creation of a new segment (case 1).

We use the following notation for replicated attributes. Let S be a shared segment whose root node is labeled by the nonterminal symbol X, and $S_x^1, \ldots, S_x^n$ the n shared-inclusion sites of the shared segment S. Each input attribute a associated with X is replicated n times, as are all other attributes in S that depend on S's input attributes. We denote the i^{th} copy of the attribute a corresponding to the i^{th} shared-inclusion site S_x^i by a^i. We call an attribute that is replicated n times an *n-replicated* attribute.

When there are no shared-inclusion sites of a shared segment S, the input attributes of S are defined by their completing semantic equations. In this case, there would be one copy of each input attribute of S and of any attribute that depends on S's input attributes. This copy is denoted by a^0. In the remainder of this subsection we assume that all replicated attributes have an additional copy corresponding to the default value.

Case 1, Creation of a New Shared Segment: When segment S is first created, all the attribute instances decorating the derivation tree for S have Null values. (This is true whether S is shared or not.) The evaluation algorithm for a shared segment must replicate the input attributes of S as well as all other attributes in S that depend on them, and evaluate all the attributes in the segment. A copy of each input attribute of S is created for each shared-inclusion site as determined by the segment linkage operation. Replication and evaluation of attributes is performed in the same way as for a subtree replacement as described in Case 2 below, with the root of the subtree in this case being the root of the segment S.

Case 2, Replacement of a Subtree in a Shared Segment: When a subtree S is replaced by a subtree S' within a shared segment, replication of attributes associated with nodes in S' is interleaved with the evaluation of attributes affected by the subtree replacement. Let r and r' be the root nodes of subtrees S and S' respectively. (These two nodes must be labeled with the same nonterminal symbol). Recall from chapter 2 that the subtree replacement operation retains the synthesized attributes of r and the inherited attributes of r'.[27] If an inherited attribute of r was replicated n times, then n

[27]The reason for this, as stated in chapter 2, is to limit the initial set of inconsistent attributes to those associated with the root of the replaced subtree.

copies of the corresponding inherited attribute of r' are created, and the values of all copies are initialized to Null.

As described in chapters 2 and 3, the dynamic attribute evaluation algorithm uses a scheduling graph — the model — to represent the dependencies among the attributes in the shared segment S that need to be reevaluated. When an n-replicated attribute instance b is scheduled for evaluation, it is evaluated once for each copy b^i of the attribute, where $1 \leq i \leq n$. The evaluation of the i^{th} copy b^i must employ the correct corresponding copy a^i of each replicated argument a. By definition, at least one argument of a replicated attribute instance must be replicated.

If the model expands to include a successor c of an n-replicated attribute b, and c is not replicated,[28] then n copies of the attribute c are created. The value of each copy of c is initialized to Null. Note that even if the attribute c had some other value before the subtree replacement, all copies of c must be reevaluated since the old value of c corresponded to a non-replicated b.

Case 3, Creation of a Use of a Shared Segment: Let S_X^{n+1} denote the new shared-inclusion site of S. Another copy a^{n+1} of each input attribute a of S is created, initialized to Null. Then, a variation of the replication/evaluation algorithm for a subtree replacement described in case 2 above is performed, with the root of S corresponding to the root of the replaced subtree.

The initial set of inconsistent attributes are the $(n + 1)^{\text{st}}$ copy of the input attributes of S (*i.e.*, the inherited attributes of X). The definition of the set *NeedToBeEvaluated* defined in chapter 2 is modified to contain either non-replicated attributes or individual copies of replicated attributes. This is done in order to avoid evaluating all copies of a replicated attribute unnecessarily.

When the model expands to include an n-replicated successor c of an $(n+1)$-replicated attribute b, another copy c^{n+1} of c is created and scheduled for evaluation. This

[28]This can happen if c is in S', or if the subtree replacement added a direct or transitive dependency from one of the shared segment's input attributes to c.

algorithm adds a copy corresponding to S_X^{n+1} to all attributes in S that depend on the inherited attributes of X, and evaluates each copy correctly.

Case 4, Deletion of a Use of a Shared Segment: Let S_X^j denote the deleted shared-inclusion site of a shared segment S. The a^j copy of each input attribute a of S corresponding to S_X^j is deleted. Then, a traversal of the dependency graph of S is performed, starting from each input attribute of S. When a replicated attribute b is encountered, the b^j copy corresponding to S_X^j is deleted.

6.4. Conglomerate Attributes

In this section, we present *conglomerate attributes*, which collect information from an unordered collection of segments connected by the **set-of** construct. We also describe an evaluation algorithm called *selective propagation* for conglomerate attributes so that a change in one segment is propagated to a second segment only if the latter actually uses the changed information.

We distinguish conglomerate attributes from aggregate attributes, which consist of a number of more-or-less independent components derived from a single segment. An example of an aggregate attribute is the symbol table containing the declarations within an Ada specification segment. An example of a conglomerate attribute is the symbol table containing the public declarations of all the specification segments of an Ada program.

A well-known problem with aggregate (and conglomerate) attributes is that a change to one component of the aggregate results in the reevaluation of all attributes that depend on any component of the aggregate. For instance, a new variable declaration results in reevaluation of all variable references in the scope of the changed declaration. Similarly, if conglomerate attributes are evaluated naively, then in the example shown in figure 6-1, a change to the exported variable of a *Module* segment is propagated to all modules, including those that do not import the facility.

Our definition of conglomerate attributes is based on *finite functions*, a type for aggregate attributes proposed by Hoover and Teitelbaum which, together with a modified attribute evaluation algorithm, reduces the overhead caused by aggregate attributes in a single-user environment [Hoover 86]. This is one of several mechanisms proposed in the literature to solve the aggregate problem [Johnson 83, Johnson 85, Demers 85, Horwitz 86]. We base our approach on Hoover's work because, unlike the others, it solves the problem in the single-user environment within the framework of the attribute grammar formalism.

The asynchronous nature of changes made by multiple users to a segmented program is the root of a fundamental difference between our work and that of Hoover. We use *finite relations*[29] to represent conglomerate attributes, rather than functions. The reason is that without synchronization between the programmers making changes to the different segments of the program, it cannot be guaranteed that the same component of the conglomerate attribute will not be defined simultaneously by more than one programmer. This is true in any multiple-user environment, whether running in a distributed or time-sharing system. To simplify the exposition of the new ideas in this section, we discuss only the changes to the attribute evaluation algorithm to handle attributes whose types are finite relations. Hoover's work can be applied directly to attributes whose propagation is fully contained within a segment, and the combination of his work with ours to reduce the aggregate overhead both within and among the segments is straightforward.

6.4.1. Definition of Conglomerate Attributes

A conglomerate attribute is a collection of components from various segments connected by the **set-of** construct. We introduce a new attribute type for conglomerate attributes: the *finite binary relation*. A binary relation on two sets D and R is a subset of $D \times R$, the Cartesian product of D and R. Every finite relation type declaration must specify one element of R as the **bottom** element. A binary relation is finite if and only if the set $C = \{(d,r) \mid d \in D, r \in R,$ and r is not **bottom**$\}$ is finite.

[29]Throughout this section, the term "relation" denotes the mathematical concept, and not the relations of the database world. This point is noted to distinguish our work from previous research in programming environments where the attribute grammar formalism is augmented with relational database constructs [Horwitz 86].

We refer to finite binary relations simply as finite relations, since all conglomerate values of interest are keyed lists that are binary mappings from a domain (the type of the key) to a range (the information stored for this key). For the conglomerate attribute *allexports* of the AG defined in figure 6-1, the domain D of the relation is the set of module names, and the range R is the set of symbol tables for the modules' exported facilities, needed to check consistency between the definition and uses of these facilities.

The following operations are defined on a finite relation R:

- **makeNull(*R*)**: Makes a null conglomerate value. A declaration of an attribute of finite relation type implicitly calls this operation to initialize the attribute to the null value. Typically used to initialize an empty symbol table for a new scope.
- **assign(*R*, *d*, *r*)**: Assigns $R \cup \{(d,r)\}$ to R. Typically adds a new module to the symbol table.
- **compute(*R*, *d*)**: If $(d,r) \in R$ and there is only one component in R whose key is d, returns r. If there is more than one component with the same key d, returns special value **multiple**. If $(d,r) \notin R$, returns **bottom**. Typically looks up a module name in the symbol table.

These are the only operations by which attributes of finite relation type may be manipulated. The reason for this restriction is that the set of segments that use a particular component in a conglomerate attribute is derived automatically from these operations.

Conglomerate attributes can only be associated with symbols of the grammar that derive collection of segments by the **set-of** construct. If the attribute grammar contains the production "Z ::= . . . **set-of**(X) . . . ", then a conglomerate attribute associated with grammar symbol Z is constructed by means of the **assign** operation with two synthesized attributes (one attribute for d and one for r) from each member of the set derived from the grammar symbol X. (For an example, see the semantic equation defining the conglomerate attribute *allexports* associated with the grammar symbol *Program* in figure 6-1.)

Other attributes refer to components of conglomerate attributes by means of the **compute** operation. The first argument of this operation indicates the conglomerate from which the component is to be selected. This conglomerate is usually accessed via an *upward remote reference* [Reps 89b]. An upward remote reference allows a non-local reference to an attribute of a different production *p* that necessarily occurs above the production where the reference is made in any tree derived from the grammar. The notation for upward remote references is {*id.attr*}, where *id* is the name of a grammar symbol of the production *p*, and *attr* is an attribute name associated with this symbol. For example, the operation **compute(**{Z.*a*}, *d*) returns the value *r* of the component (*d*,*r*) in the conglomerate attribute *a* associated with the non-terminal symbol Z. (For an example, see the semantic equation defining the local attribute *single_module_exports* associated with production *p3* in figure 6-1.)

6.4.2. Selective Propagation

We now extend the algorithm described in section 6.1 for evaluating the attributes of a segmented semantic tree to handle conglomerate attributes efficiently. We call the extended algorithm *selective propagation*. The increase in efficiency is achieved by means of *use-lists* that are maintained for each component of a conglomerate attribute. A component's use-list contains the names of segments that reference that component.

In each program segment, the set of references to components of conglomerate attributes are built from the **compute** operations within that segment by the evaluation algorithm, as follows. If a semantic equation contains a **compute** operation, a *demand* is placed on the component of the conglomerate identified by the second argument. It is not desirable to copy the entire conglomerate attribute to each segment that has access to this conglomerate (that is, the enclosed scopes) because a change to a component in the conglomerate would trigger an evaluation process in each segment, independent of whether the segment references the changed component or not. Instead, we keep copies of only those components actually referenced by **compute** operations within a particular segment in an attribute associated with the root of the segment. This attribute is called the *uses* set of the segment.

Thus, the *uses* set of a segment is a subset of the conglomerate attribute. For each element in *uses*, there is a list of references to attribute instances within the segment that depend on (use) that particular component. This list of pointers is used to evaluate attributes within a segment affected by a change to a conglomerate attribute, as will be described in paragraph 6.4.2.3. There is also a pointer from each attribute instance back to the *uses* set. These back pointers are required to update the *uses* set after edit operations; see algorithm of figure 6-7 in paragraph 6.4.2.1 below.

The information from each local segment's *uses* set is used to build for each unique component in the conglomerate the set of segments that should receive propagations if the value of that component changes. This is called the *used-by* set of the conglomerate component.

For the example of figure 6-1, the components of the conglomerate attribute *allexports* are the exported symbol tables of each module in the system. The *uses* set of a module *M* is the set of modules named in import statements of *M*. The *used-by* set of the component for a module *M* is the set of modules that import facilities from *M*.

Whenever one of the exported facilities of *M* changes, the change is propagated to all modules in the *used-by* set of the component of *M*. We can refine our notion of use-lists so that a *used-by* set is kept for each facility exported by *M*. This improves the efficiency of the attribute propagation algorithm even further since a change to an exported facility results in propagations only to those segments that reference the particular facility. This is accomplished in the general case by extending the finite binary relations to n-ary relations, and the key used by **assign** and **compute** to $n-1$ pre-specified fields.

We give a simple calculation to compare the efficiency of the attribute evaluation algorithm of conglomerate attributes with and without selective propagation.
Let

m =	the number of modules in the system,
e =	the average number of exported facilities per module,
i =	the average number of imported facilities per module,

$p =$ the average number of imported modules per module, and

$c =$ the average number of changes to an exported facility throughout the lifetime of the system.

If selective propagation is not used, each module would receive $m \times e \times c$ propagations. Using finite relations for the type of conglomerate attributes, which associate *used-by* sets with each module's exported symbol table, results in $p \times e \times c$ propagations per module. Note that p is usually much smaller then m. With *used-by* sets associated with each exported facility individually, this is improved even further to $i \times c$ propagations to each module.

We now describe incremental algorithms to maintain the *uses* and *used-by* sets (paragraphs 6.4.2.1 and 6.4.2.2), and to propagate changes made to the components of conglomerate attributes (paragraph 6.4.2.3) after each edit operation. This collection of algorithms form what we have been calling selective propagation.

6.4.2.1. Updating a Segment's Uses Set

A segment's *uses* set changes if (1) a new use site is added, (2) a use site is removed, or (3) the key of a use site is changed. Removing a use site occurs if either (a) the derivation tree node containing the key attribute identifying the component of that reference is deleted, or (b) the subtree decorated with the attribute instance that created the use is deleted. In our example grammar shown in figure 6-1, these correspond to the module name and the enclosing import statement, respectively. In the first case, deletion of the node containing the key leaves a *null* value for the key, so this becomes the same as changing the key of a use site (case 3). We present two algorithms for maintaining a segment's *uses* set, illustrated in figures 6-7 and 6-8 below.

The algorithm shown in figure 6-7 is a modified attribute evaluation algorithm that recognizes a new use of a conglomerate attribute, either by addition (case 1) or by change in value of the key of an already existing use (cases 2(a) and 3).

A new algorithm for deleting a subtree, shown in figure 6-8, updates the set of conglomerate components used in a segment, the segment's *uses* set. If the subtree being deleted contains a reference to a conglomerate component, that reference is

```
/* Attribute instances defined by compute(conglomerate, key) have an */
/* additional field, backptr, pointing back to the uses set of     */
/* the conglomerate component specified by key argument to compute. */

        function eval (ai: attribute instance): attribute value;
        begin
[1]         if ai is defined by compute(conglomerate, key) then
                /* Case (1): a new use site not yet */
                /* added to multiple-level uses set */
[2]             if backptr of ai = nil then
                    /* first reference to key within segment */
                    /* add entry for key to uses set */
[3]                 if key not in uses attribute at root of segment then
[4]                     entry = get_component_from_conglomerate(conglomerate, key);
[5]                     add entry to uses attribute;
[6]                     add ai to list of attribute references of entry;
[7]                     set backptr of ai to entry;
                    /* already references to same key within segment */
                    /* reuse entry for key in uses set */
[8]                 else
[9]                     entry = get_component_from_uses_set(conglomerate, key)
[10]                    add ai to list of attribute references of entry;
[11]                    set backptr of ai to entry;
[12]                fi
                /* Cases (2a) and (3): an old use site */
                /* whose key may have changed */
[13]            else
                    /* get previous entry from local conglomerate */
                    entry = follow backptr of ai;
                    /* same key */
[14]                if key = key of entry then
[15]                    do nothing;
                    /* different key */
                    /* remove from list of attribute references of previous entry */
                    /* add to list of attribute references of new entry */
[16]                else
[17]                    remove ai from list of attribute references of entry;
[18]                    set backptr of ai = nil;
[19]                    Do lines [3] - [12];
[20]                fi
[21]            fi
            /* evaluate attributes not defined by compute */
[22]        else
                . . .
            fi
        end
```

Figure 6-7: *Evaluation Algorithm for Conglomerate Attributes*

removed from the component. If the component has no more references, it is removed from the segment's *uses* set; the global evaluator is notified so that the segment's name is removed from the *used-by* set of the component in the parent segment containing the conglomerate attribute.

```
    procedure delete_subtree(r: treenode);
    begin
[1]     for each attribute instance, ai, associated with every
            treenode in subtree rooted at r, excluding r, do
[2]         if ai is defined by compute(conglomerate, key) then
                /* get entry for key and */
                /* remove attribute from list of attribute references of entry */
[3]             entry = follow backptr of ai;
[4]             remove ai from list of attribute references of entry;
                /* last reference to key within segment */
[5]             if list of attribute references of entry = nil then
[6]                 remove entry from uses attribute;
[7]                 remove_use_from_conglomerate(conglomerate, key);
                fi
            fi
        od

        /* free storage taken up by r */
        . . .

    end
```

Figure 6-8: *Subtree Deletion Algorithm for Conglomerate Attributes*

6.4.2.2. Change to Component's Used-by Set

The *used-by* set for each conglomerate component, indicating which segments use a particular component of a conglomerate attribute, is affected by the two functions *get_component_from_conglomerate(conglomerate, key)* and *remove_use_from_conglomerate(conglomerate, key)* invoked in the algorithms of figures 6-7 and 6-8 above. The former function adds the name of the segment that issued the call to the *used-by* set of the component whose key is specified. The latter removes the segment name from the *used-by* set. The name of the segment is not a parameter to these functions since each segment's local evaluator communicates with the global evaluator over a unique channel, which serves to identify the segment.

There are two situations that require special handling: (1) the key specifies a multiply defined component, or (2) the key specifies an undefined component. If the call to *get_component_from_conglomerate* specifies a multiply defined component, then the segment name is added to the *used-by* set of <u>any</u> component with the specified key. Multiply defined components, as well as the program segments that define them, are treated as erroneous — there is no propagation from duplicate segments to adjoining segments unless additional information is given (by the programmer) indicating which one of the duplicate segments should form part of the program being developed. (Checking for duplicate segments is elaborated in subsection 6.4.3.) We describe below in paragraph 6.4.2.3 how to handle the deletion of a component such that the correct action is taken when a key that was multiply defined becomes unique. If *remove_use_from_conglomerate* specified a multiply defined key, then the *used-by* set of each component with that key must be searched to delete the segment that invoked the function.

If *get_component_from_conglomerate* specifies a key that is not defined in the conglomerate, **bottom** is returned. A component is added to the global conglomerate with the specified key and the value **bottom**, and a *used-by* set for it is created. This is necessary to handle the correct propagations if a component with that key is defined later on. We mark such components as ''demanded-but-undefined'', and distinguish them from regularly defined components.

6.4.2.3. Propagation after Change to Conglomerate Attribute

A conglomerate attribute is changed when arguments to the **assign** operator used to construct it are modified. Recall that the **assign** operator adds a component (d,r) to a conglomerate attribute a from each segment in the collection of segments specified by the **set-of** construct, where d and r represent the key and value of the component, respectively. A change to the attributes d or r of a segment results in a change to a component of the conglomerate attribute instance a, which in turn causes propagations to affected segments.

The following algorithms handle changes in the definitions of conglomerate components.

1. Change from *r* to *r'* in segment — The component (*d*, *r'*) is transmitted from the segment where the change occurred to the parent segment where the conglomerate is defined. The global evaluator propagates the component with the changed value to segments that use that component — listed in the component's *used-by* set — to update the corresponding component of the affected segments' *uses* attribute. In the example AG shown in figure 6-1, this arises when the exports list of a module is modified.

   When a component of the *uses* attribute of a segment is changed, evaluation processes are initiated at all nodes within the segment whose associated attributes use that particular component. These nodes are determined from the list of references to attribute instances associated with the changed component in the *uses* set. These evaluation processes can proceed concurrently until they collide, when they are merged as we described in chapter 3. Or the merging could be forced to occur immediately by initializing a single model (and therefore a single evaluation process) to cover all affected nodes.

2. Definition of new component — This happens when a new key is defined, *i.e.*, a new segment is created.

   - If the key is already in a defined component of that conglomerate attribute, the value **multiple** is propagated to all segments that use that key.

   - If there is a demanded-but-undefined component with the same key as the newly defined component, then the component is marked as defined. The value of the newly defined component is propagated to all reference sites as indicated by the component's *used-by* set.

   - If no component with the specified key exists, then the component is added to the conglomerate attribute, with its *used-by* set initialized to empty.

3. Deletion of component from conglomerate attribute — This happens either because (a) the segment corresponding to the component is deleted from the program, or (b) the part of the segment defining the key attribute is deleted. In the example AG of figure 6-1, a component with key *M* is removed from the *allexports* conglomerate either if the segment containing module *M* is deleted, or if the subtree defining the name of the module *M* is deleted from the segment containing module *M*.

- If this was a duplicate component, the *used-by* set for the deleted component is combined with the *used-by* set of another component with the same key. Then, the component is deleted. If only one component is left with the key of the deleted component, the value of the remaining component is propagated to the segments on its *used-by* set. This is appropriate, for example, when all but one instance of a multiply defined module are removed.
- If the component was not a duplicate, then the component is marked as demanded-but-undefined. The *r* value is changed to **bottom**, and is propagated to all segments on the component's *used-by* set.

6.4.3. The *isUnique* Operator

In our original formulation of conglomerate attributes, reported in [Micallef 88], the **assign** operator had a secondary function: to define an *error* attribute for segments contributing components with the same key to the conglomerate. This method of checking for duplicate segments in an unordered collection is problematic when multiple conglomerate attributes are defined on the same set of segments. In this case, the check for duplicate segment names is repeated for each conglomerate since the **assign** operation is the only available operation for constructing conglomerate attributes.[30] Not only does this cause unnecessary attribute evaluation and propagation, but it also results in multiple semantic equations defining the error attribute that indicates duplicate segment names.

In the definition of conglomerate attributes presented in this thesis, we decoupled the two functions originally provided by the **assign** operator. The **assign** operator now only adds components to conglomerate attributes. A different built-in operator, **isUnique**, is used to check for duplicate segments in a program.

The **isUnique** operator is a predicate that can be used in semantic equations associated with a production defining an unordered collection of segments, such as ''*p*: *Z* ::= . . . **set-of**(*X*) . . . ''. This operator takes one argument, an attribute associated with the

[30]Multiple conglomerate attributes are required to perform inter-module semantic analysis in Ada [Micallef 90].

nonterminal symbol specified in the **set-of** construct (such as *X* in production *p*). This key attribute usually contains the name of the program unit represented by the segment derived from the nonterminal symbol.

The **isUnique** operator is invoked from within the body of the iteration operator, with the key attribute of each segment in the collection. For each segment, it returns *true* or *false*, depending on whether or not the specified key is unique. Figure 6-9 shows the semantic equation that checks that the names of library units (represented by specification segments) in an Ada program are distinct. If the name of a library unit is not distinct, then the *error* attribute for the segment representing that library unit is set to the string "<-- duplicate library unit". As was described in section 6.1, the global evaluator propagates any changes to a segment's input attributes, such as the *error* attribute, to the segment in question.

```
specification_segment:   { synthesized attributes:        name;
                           inherited attributes:          error default ""; }


p1:  ada_program ::= set-of (specification_segment);
                     { for each linked specification_segment$i
                             specification_segment$i.error =
                                     isUnique (specification_segment$i.name)
                                             ? ""
                                             : "<-- duplicate library unit"; }
```

Figure 6-9: *Checking for Library Units with Duplicate Names in Ada*

Changes to the key attribute of a segment used as an argument to the **isUnique** operator are handled incrementally, similarly to what was described for handling changes to the components of a conglomerate attribute in paragraph 6.4.2.3. A local conglomerate attribute, called *segmentNames*, is implicitly declared for each production with an associated semantic equation containing the **isUnique** operator. A component of *segmentNames* is defined for each segment in the collection, where the key of the component is equal to the key attribute specified in the **isUnique** operator, and the value of the component is Null.

Only two kinds of changes to components of the conglomerate *segmentNames* are important. If a new component is added to the conglomerate with the same key as an existing component, then the **isUnique** operator returns *false* for both segments. In the AG fragment of figure 6-9, this would cause the *error* attribute instances associated with both specification segments to be set to the string "<--duplicate library unit". If a component is deleted from *segmentNames* so that a remaining component that was multiply defined becomes unique, the **isUnique** operator returns *true* for the unique segment, causing the attribute defined by the **isUnique** operator to be reevaluated.

Chapter 7
Summarizable Attribute Grammars

An anomaly may arise during attribute evaluation of a segmented derivation tree where certain attributes in a segment may remain inconsistently attributed for a considerable amount of time. To illustrate how this may happen, consider the segmented derivation tree shown in figure 7-1. There are two segments *R* and *S*, (logically) connected at interface node *X*. There are two attributes associated with the interface node *X*, *a* and *b*, and there is a transitive dependency from *a* to *b* that goes through attributes in segment *S*. Suppose that a subtree replacement in segment *R* causes the value of attribute *a* to change. Then, all the attributes in the chain between *a* and *b* in segment *S* have to be evaluated before attribute *b* is updated. In the meantime, the attribute *b* and all of its dependent attributes in segment *R* are inconsistent. In the multi-user editor application, if the time period during which a segment has inconsistent attributes is long — which is likely to happen when segments reside on different workstations — the user ''sees'' this intermediate state of attribute evaluation and receives the wrong feedback about his change. We would therefore like to avoid such scenarios.

In general, a segmentable attribute grammar exhibits the anomaly just described if there is a direct or transitive dependency between two attributes *a* and *b* of a distributable symbol *X*, and *a* and *b* are defined in different segments (*i.e.*, one of the attributes is defined in the segment where *X* is a leaf node and the other is defined in the segment derived from *X*). Thus, the dependencies that cause the anomalous behavior must be from an inherited to a synthesized *X*, or from a synthesized to an inherited attribute of *X*.

The desirable solution to avoid this anomaly from arising is to rewrite the attribute grammar to avoid such dependencies across segments. This is always possible, in theory, since attribute grammars that use both inherited and synthesized attributes are

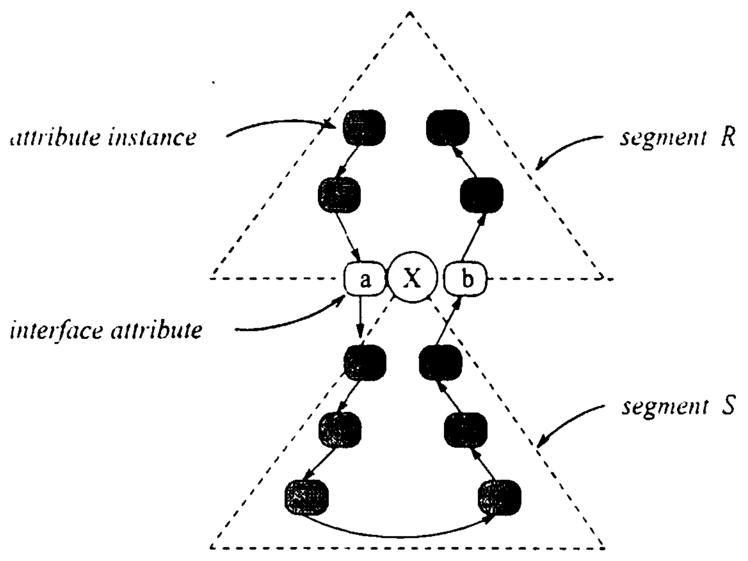


Figure 7-1: *Attribute Evaluation Anomaly in Segmented Derivation Tree*

no more powerful than methods that use just synthesized attributes [Knuth 68]. Thus, every attribute grammar can be rewritten to use only synthesized attributes and thus avoid the anomalous dependencies across segments altogether. However, writing an equivalent attribute grammar using no inherited attributes is often considerably harder, and the resulting grammar is more complicated, and more difficult to understand and change. And automatically transforming an AG that has anomalous dependencies across segments to an equivalent one that has no such dependencies is an undecidable problem.[31]

We define a subclass of the segmentable attribute grammars, called *summarizable* AGs, which do not exhibit anomalous behavior. A segmentable attribute grammar, AG_1, is summarizable if either of the following two conditions hold:

- **Condition 1:** For each distributable symbol X in AG_1, there are no direct or transitive dependencies from an inherited interface attribute of X to a synthesized interface attribute of X, or from a synthesized interface attribute of X to an inherited attribute of X.

[31]Since AGs are equivalent in power to Turing machines, checking whether two AGs compute the same function is undecidable [Cutland 80].

- **Condition 2:** AG_1 can be transformed into an equivalent attribute grammar for which condition 1 holds by one of the transformation methods to be described in the following sections.

We identify two patterns of attribute dependencies across segments commonly found in AGs defining the static semantics of programming languages that do not satisfy condition 1 stated above. The first pattern involves a direct dependency from a synthesized (inherited) interface attribute of X to an inherited (synthesized) interface attribute of X. This pattern can be automatically detected from the attribute grammar by checking the dependencies in the two productions that apply at X.

The second pattern involves a transitive dependency from a synthesized (inherited) interface attribute of X to an inherited (synthesized) interface attribute of X, where both attributes are aggregates, and the inherited (synthesized) attribute is defined to be equal to the synthesized (inherited) attribute plus some additional components. Although this pattern cannot be detected automatically — it requires knowledge of the meaning of a semantic function — a human can easily identify instances of it in AG specifications.

In the following sections, we give representative examples of these two patterns of attribute dependencies across segments, and present algorithms to transform attribute grammars that contain these dependency patterns into equivalent AGs without dependencies across segments.

7.1. Transformation involving Direct Dependencies

The first pattern is exemplified by the AG fragment for procedures in a Pascal-like language shown in figure 7-2. Each top-level procedure in the program is defined in a separate segment. A procedure consists of the procedure name, a list of formal parameters, and a sequence of statements.

The AG of figure 7-2 adds a symbol table entry for each top-level procedure defined in the program. A pair of attributes, *defs_in* and *defs_out*, are used to accomplish this; these attributes are associated with the two nonterminals, *proc_list* and *proc*. The attribute *defs_in* represents the symbol table available to a procedure and *defs_out* represents the symbol table after the procedure's header has been added. *Defs_in* is initialized to contain the symbol table entries for the global types and variables that are defined in the program before the top-level procedures.

```
    proc_list, proc:
                  { inherited attributes:              defs_in;
                    synthesized attributes:            defs_out; }

p1: program ::= ... proc_list ...
                      { proc_list.defs_in = program.sym_tab;
                        ...;}

p2: proc_list ::= /* empty */
                      { proc_list.defs_out = proc_list.defs_in; }

p3:            | proc proc_list
                      { proc.defs_in = proc_list$1.defs_in;
                        proc_list$2.defs_in = proc.defs_out;
                        proc_list$1.defs_out = proc_list$2.defs_out; }

    distributable proc;
p4: proc ::= proc_name formals stmt_seq
                      { proc.defs_out = AddEntry(proc_name.name, formals.out, proc.defs_in);
                        formals.env_in = proc.defs_out;
                        stmt_seq. env = formals.env_out;
                        ...; }
```

Figure 7-2: *AG with Direct Dependencies across Segments*

The entry for a top-level procedure is added to the symbol table in the first semantic equation associated with production *p4* by the user-defined function *AddEntry*. This function adds the name of the procedure and the signature of the procedure's formal parameters to the symbol table. (We omit the semantic equations defining the attributes *proc_name.name* and *formals.out*, which contain the procedure's name and the signatures of the formal parameters, respectively.)

The nonterminal *proc* is distributable, that is, it is an interface node between two segments. As was described in chapter 6, section 6.1, the interface attributes associated with *proc* (*defs_in* and *defs_out*) are duplicated in the parent and child segments. The value of the inherited attribute *defs_in* is computed in the parent segment, and the value of the synthesized attribute *defs_out* is computed in the child segment. When either attribute changes in value, the global evaluator propagates the change from one segment to the other.

There is a direct dependency between the attributes *defs_in* and *defs_out* associated with the nonterminal *proc* resulting from the first semantic equation associated with production *p4*. If the attribute *defs_in* associated with *proc* changes due to a modification to a global type, variable or another procedure defined before this procedure, the value of attribute *defs_out* must be recomputed. Since this attribute is synthesized, it is recomputed in the child segment and then propagated to the parent segment. Thus, the new value for *defs_out* and any attributes dependent on it in the parent segment are not available until communication with the child segment has occurred.

```
  proc:       { inherited attributes:              defs_in;
                synthesized attributes:            proc_name, formals_types; }


p3:  proc_list ::= proc proc_list
                  { proc.defs_in = proc_list$1.defs_in;
                    proc_list$2.defs_in = AddEntry(proc.proc_name, proc.formals_types,
                                                   proc.defs_in);
                    proc_list$1.defs_out = proc_list$2.defs_out; }

     distributable proc;
p4:  proc ::= proc_name formals stmt_seq
                  { proc.proc_name = proc_name.name;
                    proc.formals_types = formals.out;
                    formals.env_in = AddEntry(proc_name.name, formals.out, proc.defs_in);
                    stmt_seq. env = formals.env_out;
                    . . . ; }
```

Figure 7-3: *AG of Figure 7-2 without Direct Dependencies across Segments*

This attribute grammar can be rewritten so that after a change to the value *defs_in* associated with *proc*, the value of *defs_out* in the parent can be recomputed without going through the child segment. The resulting summarizable AG is shown in figure 7-3. The attribute *defs_out* associated with the distributable nonterminal *proc* is eliminated by duplicating its computation in both parent and child segments, as explained below. Instead, two other synthesized (interface) attributes, *proc_name* and *formals_types*, are declared for the nonterminal *proc*. This means that if the procedure's name or the signature of a formal parameter changes, the new value for the corresponding attribute is propagated to the parent.

In the original grammar, the attribute *proc.defs_out* was used in the parent segment to define the attribute *defs_in* associated with the second occurrence of *proc_list* in production *p3*. In the child segment, the attribute *proc.defs_out* was used to define *formals.env_in*. Eliminating the attribute *defs_out* to remove the direct dependency between *defs_in* and *defs_out* associated with the distributable symbol *proc* requires that the semantic equation defining *defs_out* be duplicated in every place where this attribute was previously used, in both parent and child segments. This is done in the two semantic equations defining *proc_list$2.defs_in* and *formals.env_in* in productions *p3* and *p4*, respectively.

We now give a general solution for removing direct dependencies across segments. We consider two dual cases: (1) when there is a direct dependency from an inherited attribute to a synthesized attribute of a distributable nonterminal symbol, and (2) when there is a direct dependency from a synthesized attribute to an inherited attribute of a distributable symbol.

7.1.1. Case 1: Direct Dependency from an Inherited to a Synthesized Attribute.

Let X_0 denote the distributable symbol, and productions p and q the productions in the parent and child segments that apply at X_0, defined as follows:

$$p\colon \ldots \rightarrow \ldots X_0 \ldots$$
$$q\colon X_0 \rightarrow X_1 \ldots X_n$$

Let a_{syn} and a_{inh} denote synthesized and inherited attributes associated with the distributable symbol X_0. Then, there is a direct dependency from a_{inh} to a_{syn} if a_{syn} is defined by the following semantic equation associated with production q:

$$X_0.a_{syn} = f(X_0.a_{inh}, Y_1.a_1, \ldots, Y_k.a_k)$$

where f is a semantic function, $Y_i \in \{X_0, \ldots, X_n\}$ for $1 \le i \le k$, and $Y_i.a_i$ is an attribute associated with the symbol X_j corresponding to Y_i for $1 \le i \le k$ and $0 \le j \le n$.[32]

[32]Although our presentation is specific to the case where the attribute $X_0.a_{inh}$ is the first argument to the semantic function f, the algorithms discussed below apply for any permutation of the parameters to f.

To remove the direct dependency from $X_0.a_{inh}$ to $X_0.a_{syn}$, the AG is changed as follows:

1. For each attribute $Y_i.a_i$, $1 \leq i \leq k$, such that $Y_i \neq X_0$,
   a. Declare $X_0.a_i$ to be a new synthesized interface attribute.
   b. Define each newly declared attribute from step (a) by a copy rule associated with production q:
      $X_0.a_i = Y_i.a_i$
2. In semantic equations associated with production p, replace every occurrence of $X_0.a_{syn}$ on the right-hand side of the semantic equation by $f(X_0.a_{inh}, X_0.a_1, \ldots, X_0.a_k)$.
3. In semantic equations associated with production q, replace every occurrence of $X_0.a_{syn}$ on the right-hand side of the semantic equation by $f(X_0.a_{inh}, Y_1.a_1, \ldots, Y_k.a_k)$, *i.e.*, the original semantic function defining $X_0.a_{syn}$.
4. Delete the declaration of the synthesized interface attribute a_{syn} associated with the distributable symbol X_0.

7.1.2. Case 2: Direct dependency from a synthesized to an inherited attribute.

Let X_m denote the distributable symbol, and productions p and q the productions in the parent and child segments that apply at X_m, defined as follows:

$p: X_0 \rightarrow X_1 \ \ldots \ X_m \ \ldots \ X_n$
$q: X_m \rightarrow \ \ldots$

Let a_{syn} and a_{inh} denote synthesized and inherited attributes associated with the distributable symbol X_m. Then, there is a direct dependency from a_{syn} to a_{inh} if a_{inh} is defined by the following semantic equation associated with production p:

$$X_m.a_{inh} = f(X_m.a_{syn}, Y_1.a_1, \ldots, Y_k.a_k)$$

where f is a semantic function, $Y_i \in \{X_0, \ldots, X_n\}$ for $1 \leq i \leq k$, and $Y_i.a_i$ is an attribute associated with the symbol X_j corresponding to Y_i for $1 \leq i \leq k$ and $0 \leq j \leq n$.

To remove the direct dependency from $X_m.a_{syn}$ to $X_m.a_{inh}$, the AG is changed as follows:

1. For each attribute $Y_i.a_i$, $1 \leq i \leq k$, such that $Y_i \neq X_m$,
   a. Declare $X_m.a_i$ to be an inherited interface attribute.

b. Define each newly declared attribute from step (a) by a copy rule associated with production p:

$$X_m.a_i = Y_i.a_i$$

2. In semantic equations associated with production p, replace every occurrence of $X_m.a_{inh}$ on the right-hand side of the semantic equation by $f(X_m.a_{syn}, Y_1.a_1, \ldots, Y_k.a_k)$, *i.e.*, the original semantic function defining $X_m.a_{inh}$.
3. In semantic equations associated with production q, replace every occurrence of $X_m.a_{inh}$ on the right-hand side of the semantic equation by $f(X_m.a_{syn}, X_m.a_1, \ldots, X_m.a_k)$.
4. Delete the declaration of the inherited interface attribute a_{inh} associated with the distributable symbol X_m.

7.2. Transformation involving Transitive Dependencies

A segmentable AG with transitive dependencies across segments is shown in figure 7-4. This AG defines a list of declarations, where a sublist of declarations forms a segment. There is a transitive dependency from the inherited attribute *sym_tab_in* associated with the distributable symbol *decls_segment* to the synthesized attribute *sym_tab_out* associated with the same symbol. *Sym_tab_in* contains the symbol table entries for identifiers declared prior to the nonterminal symbol that this attribute is associated with. *Sym_tab_out* contains the symbol table entries for identifiers declared prior to and including the nonterminal symbol that this attribute is associated with.

Figure 7-5 shows a functionally equivalent attribute grammar, that has no transitive dependencies across segments. The transformation is again accomplished by removing one of the interface attributes involved in the transitive dependency, and defining additional attributes to fill in the role previously assigned to the deleted attribute. The interface attribute that is eliminated is *sym_tab_out* associated with *decls_segment*. Instead of this attribute, a synthesized interface attribute *subtree_sym_tab_out* is declared for *decls_segment*; this attribute contains the symbol table for declarations defined in the segment derived from the symbol *decls_segment*. The union of this new attribute and the attribute *sym_tab_in* of *decls_segment* is equal to the value of the previous interface attribute *sym_tab_out* of *decls_segment*. Thus, the union of the two

```
decls_list, decls_segment, decls, decl:
                    { inherited attributes:            sym_tab_in;
                      synthesized attributes:          sym_tab_out; }

p1:  decls_list ::= /* empty */
                    { decls_list.sym_tab_out = decls_list.sym_tab_in; }

p2:             | decls_segment decls_list
                    { decls_segment.sym_tab_in = decls_list$1.sym_tab_in;
                      decls_list$2.sym_tab_in = decls_segment.sym_tab_out;
                      decls_list$1.sym_tab_out = decls_list$2.sym_tab_out; }

     distributable decls_segment;
p3:  decls_segment ::= decls;
                    { decls.sym_tab_in = decls_segment.sym_tab_in;
                      decls_segment.sym_tab_out = decls.sym_tab_out; }

p4:  decls ::= /* empty */
                    { decls.sym_tab_out = decls.sym_tab_in; }

p5:             | decl decls
                    { decl.sym_tab_in = decls$1.sym_tab_in;
                      decls$2.sym_tab_in = decl.sym_tab_out;
                      decls$1.sym_tab_out = decls$2.sym_tab_out; }

p6:  decl ::= var_name type_denoter
                    { decl.sym_tab_out = AddEntry(var_name.name, type_denoter.type,
                                                  decl.sym_tab_in);
                      . . . ; }
```

Figure 7-4: *AG with Transitive Dependencies across Segments*

attributes replaces all occurrences of the eliminated attribute *decls_segment.sym_tab_out* on the right-hand side of semantic equations in the parent segment (the second semantic equation associated with *p2* in figure 7-5).

The value of the synthesized interface attribute *subtree_sym_tab_out* is computed in the child segment derived from *decls_segment.* This is accomplished by defining an additional pair of attributes for the nonterminals *decls* and *decl.* This pair of attributes, *subtree_sym_tab_in* and *subtree_sym_tab_out*, play the same role as the attributes *sym_tab_in* and *sym_tab_out* associated with the same nonterminals, except that the former pair only contain declarations found in the child segment. Therefore, the attribute *subtree_sym_tab_in* associated with the topmost occurrence of *decls* in the child segment is initialized to NullList (in the second semantic equation associated with production *p3* of figure 7-5). The semantic equations defining *subtree_sym_tab_in* and *subtree_sym_tab_out* are identical to those defining the attributes *sym_tab_in* and *sym_tab_out*, except that occurrences of the latter pair of attributes on the right-hand side of the equations are replaced with the corresponding attribute from the former pair.

```
    decls_list:        { inherited attributes:          sym_tab_in;
                         synthesized attributes:        sym_tab_out; }

    decls_segment:     { inherited attributes:          sym_tab_in;
                         synthesized attributes:        subtree_sym_tab_out; }


    decls, decl:       { inherited attributes:          sym_tab_in, subtree_sym_tab_in;
                         synthesized attributes:        sym_tab_out, subtree_sym_tab_out; }

p1:  decls_list ::= /* empty */
                       { decls_list.sym_tab_out = decls_list.sym_tab_in; }

p2:             | decls_segment decls_list
                       { decls_segment.sym_tab_in = decls_list$1.sym_tab_in;
                         decls_list$2.sym_tab_in = Union(decls_segment.sym_tab_in,
                                                   decls_segment.subtree_sym_tab_out);
                         decls_list$1.sym_tab_out = decls_list$2.sym_tab_out; }

     distributable decls_segment;
p3:  decls_segment ::= decls;
                       { decls.sym_tab_in = decls_segment.sym_tab_in;
                         decls.subtree_sym_tab_in = NullList();
                         decls_segment.subtree_sym_tab_out = decls.subtree_sym_tab_out; }

p4:  decls ::= /* empty */
                       { decls.sym_tab_out = decls.sym_tab_in;
                         decls.subtree_sym_tab_out = decls.subtree_sym_tab_in; }

p5:             | decl decls
                       { decl.sym_tab_in = decls$1.sym_tab_in;
                         decls$2.sym_tab_in = decl.sym_tab_out;
                         decls$1.sym_tab_out = decls$2.sym_tab_out;
                         decl.subtree_sym_tab_in = decls$1.subtree_sym_tab_in;
                         decls$2.subtree_sym_tab_in = decl.subtree_sym_tab_out;
                         decls$1.subtree_sym_tab_out = decls$2.subtree_sym_tab_out; }

p6:  decl ::= var_name type_denoter
                       { decl.sym_tab_out = AddEntry(var_name.name, type_denoter.type,
                                                   decl.sym_tab_in);
                         decl.subtree_sym_tab_out = AddEntry(var_name.name, type_denoter.type,
                                                   decl.subtree_sym_tab_in);
                         . . . ; }
```

Figure 7-5: *AG of Figure 7-4 without Transitive Dependencies across Segments*

7.2.1. Removing Transitive Dependency from an Inherited to a Synthesized Attribute

We now present a general solution for removing the type of transitive dependencies across segments exemplified by the AG of figure 7-4. We only consider the case where the transitive dependency is from an inherited attribute associated with a distributable symbol to a synthesized attribute of the distributable symbol. The solution for the dual case is similar.

Let X_0 denote the distributable symbol, and productions p and q the productions in the parent and child segments that apply at X_0, defined as follows:

p: $\ldots \rightarrow \ldots X_0 \ldots$

q: $X_0 \rightarrow X_1 \ldots X_n$

Let a_{syn} and a_{inh} be the synthesized and inherited attributes associated with the distributable symbol X_0 such that there is a transitive dependency from a_{inh} to a_{syn}. The type of both attributes a_{syn} and a_{inh} is a list of elements. Elements in the child segment derived from X_0 are added to the list represented by a_{inh} to form the list represented by a_{syn}.

In order to describe the algorithm for removing the transitive dependency from a_{inh} to a_{syn}, we have to characterize the structure of the derivation tree containing the path of dependency edges from a_{inh} to a_{syn}. Our characterization is intentionally simple so as not to obscure the algorithm with undue details. We recognize that there are other variations of the transitive dependency pattern that are not captured by our characterization, but the transformation algorithm can be easily extended to cover such variations.

Let Y and Z denote nonterminal symbols that have the following properties:

1. Z is directly derived from X_0; *i.e.*, Z is one of the right-hand side symbols of production q.
2. Z is a recursively defined nonterminal symbol.
3. Y is derived from Z, and is not recursively defined.
4. Y has a pair of attributes, b_{inh} and b_{syn}, which have the same type and

meaning as a_{inh} and a_{syn}, respectively. That is, $Y.b_{syn}$ is defined to be equal to $Y.b_{inh}$ plus the list element derived from Y.

5. Z has a pair of attributes, c_{inh} and c_{syn}, which have the same type and meaning as a_{inh} and a_{syn}, respectively. That is, $Z.c_{syn}$ is defined to be equal to $Z.c_{inh}$ plus the list elements in the subtree derived from Z.

The path of dependency edges from $X_0.a_{inh}$ to $X_0.a_{syn}$ has the following form:

$X_0.a_{inh}, Z.c_{inh}, (Y.b_{inh}, Y.b_{syn}, Z.c_{inh})^n, Z.c_{syn}{}^{n+1}, X_0.a_{syn}$

where $n \geq 0$.

The transitive dependency from $X_0.a_{inh}$ to $X_0.a_{syn}$ is removed by transforming the AG as follows:

1. Delete the declaration of the synthesized interface attribute a_{syn} associated with the distributable symbol X_0.

2. Declare *subtree–*a_{syn} to be a new synthesized interface attribute of X_0 with the same type as a_{syn}.

3. For the symbol Y with the properties defined above, declare an additional pair of attributes, *subtree–*b_{syn} and *subtree–*b_{inh} with the same type as b_{syn} and b_{inh}.

4. For the symbol Z with the properties defined above, declare an additional pair of attributes, *subtree–*c_{syn} and *subtree–*c_{inh} with the same type as b_{syn} and b_{inh}.

5. Let X_i be the symbol in production q: $X_0 \rightarrow X_1 \ldots X_n$ such that $X_i = Z$, $1 \leq i \leq n$. Add the following semantic equation to production q to initialize the *subtree–*c_{inh} attribute of this symbol:

   X_i*.subtree–*c_{inh} = *NullList*();

   The user-defined function *NullList* returns the data structure representing the empty list for this attribute type.

6. In productions with Y or Z on the left-hand side, add semantic equations defining the attributes Y*.subtree–*b_{syn} or Z*.subtree–*c_{syn} by copying the semantic equation defining the attribute $Y.b_{syn}$ or $Z.c_{syn}$ in the same production, replacing occurrences of $Y.b_{syn}$ and $Y.b_{inh}$ or $Z.c_{syn}$ and $Z.c_{inh}$ by Y*.subtree–*b_{syn} and Y*.subtree–*b_{inh} or Z*.subtree–*c_{syn} and Z*.subtree–*c_{inh} respectively.

7. In productions with Y or Z on the right-hand side, add semantic equations

defining the attributes $Y.subtree\text{--}b_{inh}$ or $Z.subtree\text{--}c_{inh}$ by copying the semantic equation defining the attribute $Y.b_{inh}$ or $Z.c_{inh}$ in the same production, replacing occurrences of $Y.b_{syn}$ and $Y.b_{inh}$ or $Z.c_{syn}$ and $Z.c_{inh}$ by $Y.subtree\text{--}b_{syn}$ and $Y.subtree\text{--}b_{inh}$ or $Z.subtree\text{--}c_{syn}$ and $Z.subtree\text{--}c_{inh}$ respectively.

8. Replace occurrences of $X_0.a_{syn}$ on the right-hand side of semantic equations by

   $Union(X_0.a_{inh}, X_0.subtree\text{--}a_{syn})$

   where *Union* is a user-defined semantic function that takes two lists of the same type and returns a merged list.

7.2.2. Removing Transitive Dependency for AG with Nested Segments

Transforming an AG with a transitive dependency across segments to one without such a dependency requires the introduction of an additional pair of attributes for every nonterminal symbol in the path of direct dependency edges constituting the transitive dependency. If there are nested segments of the same type (*i.e.*, labeled by the same nonterminal symbol), then the number of additional attributes that are needed to perform the transformation is unbounded. Figure 7-6 shows such an AG for specifying a list of declarations: a segment derived from the distributable symbol *decls* has an interface node to another *decls* segment if it contains an instance of production *p3*.

To remove the transitive dependency from *decls.sym_tab_in* to *decls.sym_tab_out* from each occurrence of the interface node *decls* along the lines of the solution given in the previous subsection, an additional pair of attributes for each nested *decls* segment is required. The function of this pair of attributes is to compute the one-element list containing the symbol table entry for the identifier declared within that segment. Since the nesting level of each segment is only known at run-time, the transformation of the AG would have to be performed at run-time. Although this can be done using a parameterized version of the solution we presented above, where the parameter represents the nesting level of the segment, we believe that such a segment organization is unnecessary, and the AG can be written in a way to avoid having nested segments of the same type. For instance, declaring the nonterminal *decl* distributable instead of *decls* in figure 7-6 results in an AG that has no nested segments of the same type. Yet,

```
decls, decl:        { inherited attributes:          sym_tab_in;
                      synthesized attributes:         sym_tab_out; }

p1:   ... ::= ... decls ... ;
                    { decls.sym_tab_in = NullList; }

      distributable decls;
p2:   decls ::= /* empty */
                    { decls.sym_tab_out = decls.sym_tab_in; }

p3:            | decl decls
                    { decl.sym_tab_in = decls$1.sym_tab_in;
                      decls$2.sym_tab_in = decl.sym_tab_out;
                      decls$1.sym_tab_out = decls$2.sym_tab_out; }

p4:   decl ::= var_name type_denoter
                    { decl.sym_tab_out = AddEntry(var_name.name, type_denoter.type,
                                                  decl.sym_tab_in; }
                      ... ; }
```

Figure 7-6: *AG with Nested Segments of the Same Type*

each segment in the resulting AG derives the same set of strings that the AG in figure 7-6 does. As we described in chapter 5, section 5.3, segmentation schemes allowing optional segments and list segments can also be written without having nested segments of the same type.

7.3. Related Work

As far as we know, no other researchers have considered the extensions to context-free grammars and Knuth's original attribute grammar formalism necessary to express complex inter-module syntactic and semantic connections, respectively.

Perry's Inscape system [Perry 89] also unifies the specification of inter-module and intra-module semantics. A set of preconditions, postconditions and obligations is associated with every statement in a subroutine, with each subroutine, and with each module interface (specified as a list of exported subroutines). A set of invariants are associated with every global variable and type definition. Semantic analysis is accomplished by propagation of statement preconditions to their ceilings, the earliest points in a subroutine after which they are not invalidated, and by propagation of statement postconditions and obligations to their floors, the latest points in the subroutine before which they are not invalidated. All preconditions and obligations must be satisfied by corresponding postconditions. If there is no invalidation within the subroutine, then preconditions, postconditions and obligations are propagated to the interface to the subroutine; a subset of them, selected by the programmer, is propagated to the interface of the module. Since full theorem-proving is infeasible, Inscape performs its analysis using only simple symbol manipulation, and thus the ''correctness'' guaranteed cannot depend on deep properties and implications of the predicates that appear in the preconditions, postconditions and obligations.

This approach is orthogonal to attribute grammars and attribute evaluation techniques, since there is nothing about attribute grammars that limits them to the traditional symbol resolution, type checking and code generation, and in fact we believe the semantics analysis described could be implemented using our extended attribute grammars formalism. The main difficulty would be representation of the preconditions, postconditions, obligations and invariants in terms of aggregate attributes. Teitelbaum and Chapman's recent work on higher-order attribute grammars [Teitelbaum 90], where the attribute grammar can describe updates to the derivation tree as well as to the attributes, may help since then the logical clauses could be represented as part of the derivation tree rather than as attributes. This representation is currently used in the Inscape implementation, which was constructed using the Gandalf system [Habermann 86], where action routines rather than attribute grammars are used to express semantics processing.

More recent work on the Gandalf system describes a model for scaling up the system to support large software databases and multiple users [Krueger 88]. They allow multiple (context-free) grammars to describe the database organization, and segmentation of the database at grammar boundaries. Segmentation is essential for two reasons: (1) to support large databases, so that the entire database does not have to be loaded into a user process space, and (2) to support multiple users, so concurrency control can be applied at the segment level. Multiple grammars are combined at *segment nodes*; a segment node is a terminal symbol representing an abstraction in one grammar, and the start symbol in the grammar defining the abstraction. Segment nodes are similar to our distributable nonterminal symbols. The scaled up version of the Gandalf system still uses action routines for semantics processing.

Boehm and Zwaenepoel [Boehm 87] describe a distributed algorithm for parallel attribute evaluation, in order to speed up the compilation process, but their approach seems to work only for monolithic programs rather than what would be separately compiled modules in a conventional compilation system. The parse tree is divided into subtrees, which are evaluated in parallel by evaluators executing on different machines. The attribute grammar specifies at which nonterminals the parse tree may be split, and the minimum size of the subtree to be evaluated separately. The attribute grammar is based on a conventional context-free grammar, and the subtrees that are evaluated in parallel do not correspond to modular units of the language. Their distributed evaluator differs from our combined local and global evaluator because: (1) it is not incremental, that is, it performs a complete evaluation of all attribute instances in a tree; (2) it uses a built-in evaluation strategy — bottom subtrees are evaluated using a static strategy while other attribute instances are evaluated dynamically; (3) it has no support for programming-in-the-large constructs.

Klaiber and Gokhale also describe parallel non-incremental evaluators for attribute grammars, but in their case for execution on a multiprocessor [Klaiber 89]. The class of grammars handled by their parallel evaluator are the absolutely noncircular AGs (ANCAGs) [Kennedy 76]. The plans generated for ANCAGs are similar to the ones we described in chapter 4, except that they are parameterized with the set of input attributes needed to evaluate the attributes in the plan. The set of input attributes of a plan depends on the plan's context in the parse tree. Plans are parallelized by forking a process for each independent visit instruction in the plan. Weights attached to nodes of the tree estimate the cost of evaluating the subtrees rooted at the nodes, so that it is possible to determine at evaluation-time whether the cost of starting a new process for a subtree outweighs the speedup achieved by parallelization, in which case the code is executed sequentially. Because simulations showed that productions of the form $X \rightarrow Y X$ severely limit the amount of possible parallelism, Klaiber and Gokhale propose a method for restructuring the attribute grammar to instead use the production $X \rightarrow Y^+$ (*i.e.*, X expands to one or more instances of Y). This allows clusters of several small subtrees rooted at Y to be executed in parallel, where otherwise each subtree would have been too small to justify the overhead of forking a new process. Although superficially this form of a list production may resemble our **set-of** construct, it still derives an ordered list, requiring that attribute dependencies among the symbols in the list flow from left to right.

Our algorithms for conglomerate attributes are based on an efficient incremental solution presented by Hoover for evaluating aggregate attributes [Hoover 86]. Hoover defines a new attribute type for aggregate attributes, the *finite function*, and primitive operators for manipulating attributes of this type. A *key tree* is maintained for each component of an attribute of finite function type, which contains non-local edges from the site in the derivation tree defining the component with the specified key to the sites which use that component. This allows efficient propagation following a change to an aggregate component. Our work on conglomerate attributes extends and differs from Hoover's work in several ways. (1) Components of a conglomerate attribute are defined in different segments of the decentralized tree, invalidating the concept of a key tree. (2) Components with the same key may be defined asynchronously by multiple users, thereby requiring the type to be a relation rather than a function. (3) A new mechanism for detecting duplicate components is needed. (4) We use our merging algorithm, which performs incremental evaluation when there are multiple inconsistent sites in a derivation tree, to efficiently propagate a changed component to attribute instances that use that component within a segment, rather than maintaining a key tree locally within the segment.

Chapter 8
Conclusion

8.1. Contributions

The research reported in this thesis significantly advances previous work on semantics-based editors that use the attribute grammar formalism in two areas: (1) incremental attribute evaluation algorithms for multiple asynchronous subtree replacements, either on a centralized tree or within a segment on a decentralized tree, and (2) extension of the classical attribute grammar formalism to allow the specification and analysis of interface consistency of large programs. These results make it possible to construct semantics-based editors for use by teams of software developers building or maintaining large software systems, whereas previously, such editors were only usable by single programmers writing small programs.

We presented a family of algorithms for performing incremental attribute evaluation when multiple asynchronous modifications are made to the program being developed or maintained. These algorithms differ in how they balance the tradeoff between algorithm efficiency and expressiveness of the attribute grammar. This is important because we anticipate that our work will be incorporated in editors for other application domains, not just programming or software development. These other applications may have different definitions of efficiency, and may impose different requirements on the expressiveness of the attribute grammar. The characteristics of the application domain can then be used to select the most efficient evaluation strategy for each particular editor.

We defined an extension of classical AGs to allow the specification of interface consistency checking for programs composed of many modules. Classical AGs can specify the static semantics of monolithic programs or modules, but not inter-module semantics; the latter was done in the past using *ad hoc* techniques. Extended AGs specify the interface static semantics of programming-in-the-large constructs found in real programming languages, where a program may be composed of different kinds of modules, modules may be nested arbitrarily deep to form a hierarchical program structure, and modules containing common definitions may be textually included in other modules that use those definitions. Our incremental evaluation algorithms, now applied at each module, are augmented with a global evaluator for coordinating the efforts of the modules' local evaluators, so that the combined actions of the global and local evaluators result in a complete static semantic analysis of a multi-module program.

The contributions of the thesis research are threefold: (1) a body of theoretical results regarding incremental attribute evaluation for multiple asynchronous subtree replacements, and specification of interface consistency analysis by extended attribute grammars, (2) the immediate application to multi-user semantics-based environments for software development and maintenance, to improve programmer productivity by reducing communication costs (and snafus); and (3) a foundation for other applications involving dependencies among data and changes to data.

8.2. Future Work

One promising area for applying our results is derived data in distributed objectbases. By derived data, we mean data or integrity constraints that are defined in terms of other data items in the database. The immediate application is to what the database community calls "triggers", that is, automatically recomputing derived values whenever necessary, and enforcing constraints whenever updates are made. One way to think of this application of our work is "blowing up" a derivation tree node with its children nodes and attributes to a composite object with component objects, with relationships to other objects as well as status information represented as attributes [Banerjee 87]. There has been some work in this area already, by Hudson and King in their Cactis project [Hudson 89]. Cactis applies AG evaluation algorithms to recomputation of derived data in a centralized database, but it does not support multiple users. A major difficulty in this application is that objectbases may be arbitrary graphs, while our work has dealt only with trees. One direction for future work is therefore to extend our evaluation techniques to deal with attributed graphs, taking advantage of previous work on incremental evaluation of attributed graphs for single edits [Kaplan 87, Alpern 88].

Distributed objectbases are considered by many researchers to be an appropriate basis for software development environments [Rowe 89, Neuhold 89]. Automatically maintaining consistency, or detecting inconsistency, among software artifacts in the database is essential throughout the software lifecycle as requirements, design documents, source code, test cases and so on change over time. In this thesis we have considered source code consistency defined by the static semantics of the implementation language. A second area for future study is to extend the framework for multi-user semantics-based environments developed in this thesis to support the entire software lifecycle. This entails the development of a theory of software system consistency, where consistency is defined in terms of the various software artifacts written (or generated) throughout a system's lifetime. Such a theory would allow one to define consistency between a module's source code and a set of test data when the test data is sufficient for testing the module according to some test coverage criterion. Is the attribute grammar formalism an appropriate basis for defining software system consistency? Can the evaluators developed for checking static semantic consistency of a program be used with other definitions of consistency?

A third area that requires further investigation is a more comprehensive analysis of the various attribute evaluation algorithms developed in this thesis for multiple asynchronous subtree replacements. We believe that the worst-case complexity analysis does not provide much insight into how well these algorithms perform in practice. One possibility is to study the expected-case complexity of the algorithms. In order to do this, the distribution of the inputs to the algorithms is needed. The inputs consist of an attribute grammar, which derives a (usually infinite) number of semantic trees, and sequences of asynchronous edits. Although the problem of finding the distribution is in general a hard problem, focusing on one application domain, such as programming editors, may make the problem more tractable.

Alternatively, or in conjunction with an expected-case analysis, the performance of the algorithms for specific application domains may be studied empirically or by simulation. Such a study would help answer the following questions, essential in order to transfer this technology to everyday tools used by software developers. How many software developers can be supported by the environment before response time is severely degraded? How many lines of code can the environment handle? How much network traffic is generated when modules reside on different workstations, and can the message complexity be reduced by an alternate assignment of modules to workstations?

Finally, finding a lower bound for the problem of incremental attribute evaluation for multiple asynchronous edits is still an open question. A trivial lower bound is the number of attributes affected by the multiple edits, with attributes affected by more than one edit counted once. We conjecture that there exist no algorithms with such a complexity for both the number of attributes evaluated and the bookkeeping overhead, except maybe for some overly restricted attribute grammar subclass.

Bibliography

[AdaTEC 82] AdaTEC the SIGPLAN Technical Committee on Ada.
Reference Manual for the Ada Programming Language
United States Department of Defense, 1982.
Draft Revised MIL-STD 1815.

[Alblas 90] Henk Alblas.
Concurrent Incremental Attribute Evaluation.
In *International Workshop on Attribute Grammars and their Applications*. Springer-Verlag, Paris, France, September, 1990.

[Alpern 88] Bowen Alpern, Alan Carle, Barry Rosen, Peter Sweeney, Kenneth Zadeck.
Graph Attribution as a Specification Paradigm.
In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 121-129. Boston, Massachusetts, November, 1988.

[Banerjee 87] Jay Banerjee, Haong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, Nat Ballou, and Hyoung-Joo Kim.
Data Model Issues for Object-Oriented Applications.
ACM Transactions on Office Information Systems 5(1):3-26, January, 1987.

[Bochmann 76] G.V. Bochmann.
Semantic Evaluation from Left to Right.
Communications of the ACM 19:55-62, 1976.

[Boehm 87] Hans-Juergen Boehm and Willy Zwaenepoel.
Parallel Attribute Grammar Evaluation.
September, 1987.

[Chandhok 85] Ravinder Chandhok, David B. Garlan, Dennis Goldenson, Philip L. Miller and Mark Tucker.
Programming Environments Based on Structure Editing: The GNOME Approach.
In Anthony S. Wojcik (editor), *1985 National Computer Conference*, pages 359-370. AFIPS, Chicago, IL, July, 1985.

[Cutland 80] Nigel J. Cutland.
Computability.
Cambridge University Press, Great Britain, 1980.

[Demers 81] Alan Demers, Thomas Reps and Tim Teitelbaum.
Incremental Evaluation for Attribute Grammars with Applications to Syntax-directed Editors.
In *8th Annual ACM Symposium on Principles of Programming Languages*, pages 105-116. Williamsburg VA, January, 1981.

[Demers 85] Alan Demers, Anne Rogers and Frank Kenneth Zadeck.
Attribute Propagation by Message Passing.
In *SIGPLAN 1985 Symposium on Language Issues in Programming Environments*, pages 48-59. Seattle, WA, June, 1985.
Proceedings published as *SIGPLAN Notices*, 20(7), July, 1985.

[DeRemer 76] Frank DeRemer and Hans H. Kron.
Programming-in-the-Large Versus Programming-in-the-Small.
IEEE Transactions on Software Engineering SE-2(2), June, 1976.

[Engelfriet 82] Joost Engelfriet and Gilberto File.
Simple Multi-Visit Attribute Grammars.
Journal of Computer and System Sciences 24:283-314, 1982.

[Farrow 84] Rodney Farrow.
Generating a Production Compiler from an Attribute Grammar.
IEEE Software 1(4):77-93, October, 1984.

[Feldman 79] S.I. Feldman.
Make — A Program for Maintaining Computer Programs.
Software — Practice & Experience 9(4):255-265, April, 1979.

[Ganzinger 77] Harald Ganzinger, Knut Ripken and Reinhard Wilhelm.
Automatic Generation of Optimizing Multipass Compilers.
In *Information Processing 77*, pages 535-540. North-Holland Pub. Co., New York, 1977.

[Garlan 84] David B. Garlan and Philip L. Miller.
GNOME: An Introductory Programming Environment Based on a Family of Structure Editors.
In Peter Henderson (editor), *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 65-72. Pittsburgh, April, 1984.
Proceedings published as *SIGPLAN Notices*, 19(5), May 1984 and of *Software Engineering Notes*, 9(3), May 1984.

[Geitz 87] Bob Geitz.
Asynchronous Subtree Replacement for Language-Based Editors.
1987.
Oberlin College and Cornell University.

[Greif 88] Irene Greif (editor).
Computer-Supported Cooperative Work: A Book of Readings.
Morgan Kaufman, San Mateo CA, 1988.

[Habermann 81] A. Nico Habermann and Dewayne E. Perry.
System Composition and Version Control for Ada.
Software Engineering Environments.
North-Holland, New York, 1981.

[Habermann 86] A.N. Habermann and D. Notkin.
Gandalf: Software Development Environments.
IEEE Transactions on Software Engineering SE-12(12):1117-1127, December, 1986.

[Hoover 86] Roger Hoover and Tim Teitelbaum.
Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars.
In *SIGPLAN '86 Symposium on Compiler Construction*, pages 39-50. Palo Alto, CA, June, 1986.
Proceedings published as *SIGPLAN Notices*, 21(7), July, 1986.

[Hoover 87] Roger Hoover.
Incremental Graph Evaluation.
PhD thesis, Cornell University, May, 1987.
Technical Report 87-836.

[Horwitz 86] Susan Horwitz and Tim Teitelbaum.
Generating Editing Environments Based on Relations and Attributes.
ACM Transactions on Programming Languages and Systems 8(4):577-608, October, 1986.

[Hudson 88] Scott E. Hudson and Roger King.
Semantic Feedback in the Higgens UIMS.
IEEE Transactions on Software Engineering 14(8):1188-1206, August, 1988.

[Hudson 89] Scott E. Hudson and Roger King.
Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System.
ACM Transactions on Database Systems 14(3):291-321, September, 1989.

[Jazayeri 75] M. Jazayeri, W.F. Ogden and W.C. Rounds.
The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars.
Communications of the ACM 18(12):697-706, December, 1975.

[Johnson 82] Gregory F. Johnson and Charles N. Fischer.
Non-syntactic Attribute Flow in Language Based Editors.
In *9th Annual ACM Symposium on Principles of Programming Languages*, pages 185-195. Albuquerque NM, January, 1982.

[Johnson 83] Gregory F. Johnson.
An Approach To Incremental Semantics.
PhD thesis, University of Wisconsin at Madison, 1983.

[Johnson 85] Gregory F. Johnson and C.N. Fischer.
A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors.
In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 141-151. New Orleans, Louisiana, January, 1985.

[Jourdan 89] Martin Jourdan and Didier Parigot.
The FNC-2 System User's Guide and Reference Manual
INRIA, France, 1989.

[Jourdan 90] Martin Jourdan, Didier Parigot, Catherine Julie, Olivier Durin and Carole le Bellec.
Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System.
In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 209-222. June, 1990.
Proceedings published as *SIGPLAN Notices*, 25(6), June 1990.

[Kaiser 87] Gail E. Kaiser and Simon M. Kaplan.
Reliability in Distributed Programming Environments.
In *Sixth Symposium on Reliability in Distributed Software and Database Systems*, pages 45-55. Kingsmill—Williamsburg, VA, March, 1987.

[Kaiser 90] Gail E. Kaiser and Simon M. Kaplan.
Parallel and Distributed Incremental Attribute Evaluation Algorithms for Multi-User Software Development Environments.
Technical Report CUCS-019-90, Columbia University Department of Computer Science, April, 1990.
Submitted for publication.

[Kaplan 86] Simon M. Kaplan and Gail E. Kaiser.
Incremental Attribute Evaluation in Distributed Language-Based Environments.
In *5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 121-130. Calgary, Alberta, Canada, August, 1986.

[Kaplan 87] Simon M. Kaplan.
Incremental Attribute Evaluation on Node-Label Controlled Graphs.
Technical Report UIUCDCS-R-87-1309, University of Illinois at Urbana-Champaign Department of Computer Science, May, 1987.

[Kastens 80] Uwe Kastens.
Ordered Attribute Grammars.
Acta Informatica 13:229-256, 1980.

[Kastens 82] U. Kastens, B. Hutt and E. Zimmermann.
Lecture Notes in Computer Science. Volume 141: *GAG: A Practical Compiler Generator.*
Springer-Verlag, Heidelberg, 1982.

[Kennedy 76] K. Kennedy and S.K. Warren.
Automatic Generation of Efficient Evaluators for Attribute Grammars.
In *Third Annual ACM Symposium on Principles of Programming Languages*, pages 32-49. Atlanta, GA, January, 1976.

[Kincaid 85] Christine M. Kincaid, Pierre B. Dupoint and A. Roger Kaye.
Electronic Calendars in the Office: An Assessment of User Needs and Current Technology.
ACM Transactions on Office Information Systems 3(1):89-102, January, 1985.

[Klaiber 89] Alexander Klaiber and Maya Gokhale.
Parallel Evaluation of Attribute Grammars.
In Fred Ris and Peter M. Kogge (editor), *1989 International Conference on Parallel Processing*, pages 193-201. The Pennsylvannia State University Press, University Park PA, 1989.

[Knuth 68] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2):127-145, June, 1968.

[Knuth 71] Donald E. Knuth.
Semantics of Context-Free Languages: Correction.
Mathematical Systems Theory 5(1):95-96, March, 1971.

[Krueger 88] Charles W. Krueger and Annalisa Bogliolo.
Scaling Up: Segmentation and Concurrency in Large Software Databases.
Technical Report CMU-CS-87-178, Carnegie Mellon University Department of Computer Science, February, 1988.

[Micallef 88] Josephine Micallef and Gail E. Kaiser.
Version and Configuration Control in Distributed Language-Based Environments.
In Jurgen F.H. Winkler (editor), *International Workshop on Software Version and Configuration Control*, pages 119-143. B.G. Teubner, Stuttgart, January, 1988.

[Micallef 90] Josephine Micallef and Gail E. Kaiser.
Extending the Mercury System to Support Teams of Ada Programmers.
In *1st International Symposium on Environments and Tools for Ada*, pages 1-12. Redondo Beach CA, April, 1990.
In press.

[Morris 81] Joseph M. Morris and Mayer D. Schwartz.
The Design of a Language-Oriented Editor for Block-Structured Languages.
In *SIGPLAN SIGOA Symposium on Text Manipulation*, pages 28-33. Portland OR, June, 1981.
Proceedings published as *SIGPLAN Notices*, 16(6), June 1981.

[Narayanaswamy 87]
K. Narayanaswamy and Walt Scacchi.
Maintaining Configurations of Evolving Software Systems.
IEEE Transactions on Software Engineering SE-13(3):324-334, March, 1987.

[Neal 87] Lisa Rubin Neal.
Cognition-Sensitive Design and User Modeling for Syntax-Directed Editors.
In *CHI + GI '87 Conference on Human Factors in Computing Systems and Graphics Interface*, pages 99-102. ACM, Toronto, Canada, April, 1987.

[Neuhold 89] Erich Neuhold and Michael Stonebraker (editors).
Future Directions in DBMS Research.
SIGMOD Record 18(1):17-26, March, 1989.

[Peckham 90] Stephen B. Peckham.
Incremental Attribute Evaluation and Multiple Subtree Replacements.
PhD thesis, Cornell University, February, 1990.
TR 90-1093.

[Perry 85] Dewayne E. Perry and W. Michael Evangelist.
An Empirical Study of Software Interface Errors.
In *International Symposium on New Directions in Computing*, pages 32-38. IEEE Computer Society, Trondheim, Norway, August, 1985.

[Perry 87] Dewayne E. Perry and W. Michael Evangelist.
An Empirical Study of Software Interface Faults — An Update.
In *20th Annual Hawaii International Conference on Systems Sciences*, pages 113-126. Kona HI, January, 1987.

[Perry 89] Dewayne E. Perry.
The Inscape Environment.
In *11th International Conference on Software Engineering*, pages 2-9. IEEE Computer Society, Pittsburgh PA, May, 1989.

[Reps 83] Thomas Reps, Tim Teitelbaum and Alan Demers.
Incremental Context-Dependent Analysis for Language-Based Editors.
ACM Transactions on Programming Languages and Systems 5(3):449-477, July, 1983.

[Reps 84a]	Thomas Reps and Tim Teitelbaum. The Synthesizer Generator. In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 41-48. Pittsburgh, PA, April, 1984. Proceedings published as *SIGPLAN Notices*, 19(5), May, 1984.
[Reps 84b]	Thomas Reps. *Generating Language-Based Environments.* M.I.T. Press, Cambridge, MA, 1984.
[Reps 84c]	Thomas Reps and Bowen Alpern. Interactive Proof Checking. In *11th ACM Symposium on Principles of Programming Languages*, pages 36-45. Salt Lake City, Utah, January, 1984.
[Reps 86]	T. Reps, C. Marceau and T. Teitelbaum. Remote Attribute Updating for Language-Based Editors. In *13th ACM Symposium on Principles of Programming Languages*, pages 1-13. St. Petersburg Beach, FL, January, 1986.
[Reps 88]	Thomas Reps. Incremental Evaluation for Attribute Grammars with Unrestricted Movement Between Tree Modifications. *Acta Informatica* 25:155-178, 1988.
[Reps 89a]	Thomas W. Reps and Tim Teitelbaum. *Texts and Monographs in Computer Science: The Synthesizer Generator A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1989.
[Reps 89b]	Thomas W. Reps and Tim Teitelbaum. *Texts and Monographs in Computer Science: The Synthesizer Generator Reference Manual.* Springer-Verlag, New York, 1989.
[Reps 89c]	Thomas W. Reps and Tim Teitelbaum. *Texts and Monographs in Computer Science: The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1989.
[Ross 85]	R. Ross. Design of Personal Computer Software. *Insights Into Personal Computers.* IEEE Press, New York, 1985, pages 282-300.
[Rowe 89]	Lawrence A. Rowe and Sharon Wensel (editors). 1989 ACM SIGMOD Workshop on Software CAD Databases. February, 1989.
[Sleator 83]	Daniel D. Sleator and Robert Endre Tarjan. A Data Structure for Dynamic Trees. *Journal of Computer and System Sciences* 26:362-391, 1983.

[Teitelbaum 90] Tim Teitelbaum and Richard Chapman.
Higher-Order Attribute Grammars and Editing Environments.
In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 197-208. White Plains, NY, June, 1990.
Proceedings published as *SIGPLAN Notices*, 25(6), June, 1990.

[Tichy 79] Walter F. Tichy.
Software Development Control Based on Module Interconnection.
In *4th International Conference on Software Engineering*. Munich, Germany, September, 1979.

[Uhl 82] J. Uhl, S. Drossopoulou, G. Persch, G. Goos, M. Dausmann, G. Winterstein, and W. Kirchgassner.
Lecture Notes in Computer Science 139: An Attribute Grammar for the Semantic Analysis of Ada.
Springer-Verlag, Heidelberg, FRG, 1982.

[Waite 84] W. Waite and G. Goos.
Compiler Construction.
Springer-Verlag, New York, 1984.

[Wegman 80] Mark N. Wegman.
Parsing for structural editors (Extended Abstract).
In *21st Annual Symposium on Foundations of Computer Science*, pages 320-327. Syracuse NY, October, 1980.

[Wirth 82] Niklaus Wirth.
Programming in Modula-2.
Springer-Verlag, New York, 1982.

[Wolf 85] Alexander L. Wolf, Lori A. Clarke and Jack C. Wileden.
Ada-Based Support for Programming-in-the-Large.
IEEE Software 2(2):58-71, March, 1985.

[Yeh 83] Dashing Yeh.
On Incremental Evaluation of Ordered Attribute Grammars.
BIT 23:308-320, 1983.

[Yeh 88] Dashing Yeh and Uwe Kastens.
Improvements of an Incremental Evaluation Algorithm for Ordered Attribute Grammars.
SIGPLAN Notices 23(12):45-50, December, 1988.

[Zaring 90] Alan K. Zaring.
Parallel Evaluation in Attribute Grammar-Based Systems.
PhD thesis, Cornell University, August, 1990.
Technical Report 90-1149.

Appendix A
The MERCURY System

A.1. Overview

The MERCURY system generates multi-user, distributed, semantics-based editors from an attribute grammar specification of the desired programming language. Each MERCURY editor detects syntactic and semantic inconsistencies, as defined in its language specification, with immediate feedback regarding inconsistencies. Like other semantics-based editors, such analysis is applied incrementally within each modular program unit while it is being edited by an individual editor user working in isolation. The innovative feature of MERCURY editors is that they also provide this capability among units to notify all affected users of any semantic inconsistencies introduced into their own program units by changes to the interfaces of units being modified by other users. This was illustrated in the Smod example given in the introductory chapter of this thesis.

Each MERCURY editor provides a number of levels of change propagation as desired by the individual programmers cooperating on a large software project. Each programmer editing a program unit can request that (1) his changes to the interface of his program unit be immediately propagated, at the individual editing command level of granularity, to inform any other programmers of changes that affect their units — with subsequent earliest-possible detection of newly introduced static semantic errors; or (2) that such change propagation be delayed until requested, to avoid premature communication of interface changes that are only under consideration and have not yet been committed.

Further, each programmer can also select that (3) any changes made by other programmers be immediately propagated to notify him of changes to the interfaces of other program units that adversely affect his own units, again with subsequent earliest-possible detection of newly introduced interface mismatches; or (4) that notification is delayed to avoid bombardment by error messages when many interfaces are undergoing significant changes, such as during early development stages or when it is desirable to assume the interface provided by earlier versions of other program units for an extended period of time.

MERCURY consists of two parts: (1) the editor generator, and (2) run-time support required to run the generated distributed editors. The editor generator takes as input an AG for the desired programming language, which is linked to a kernel containing algorithms common to all generated editors to produce a semantics-based editor tailored to that particular language. Copies of this editor are installed on each machine. Each invocation of one of these copies is known as a *local editor*, while the entire system involving all these editors and the run-time support is called the *distributed editor*.

MERCURY was implemented by modifying the Synthesizer Generator [Reps 89c] developed at Cornell University, which supports incremental static semantic analysis within a monolithic program in a single-user editor. MERCURY runs on a variety of Unix systems using X windows.

A.2. The Editor Generator

The current implementation of MERCURY supports only a flat segment (module) organization, that is, programs consisting of a collection of monolithic segments. The specification language used for these restricted segmentable attribute grammars is therefore considerably simpler than the one described in chapter 6:

- The start symbol of the AG represents the distributable nonterminal symbol; that is, this symbol derives a segment of the program.
- Because of the simple program structure, the root production of the program is not explicitly represented. The implicit root production for an AG whose distributable symbol is *module* would be ‘‘*program* ::= **set-of**(*module*)’’.
- Certain attributes of the distributable symbol are declared to be *interface attributes*, meaning they represent dependencies across segments (*e.g.* the

imports and exports lists of modules). This makes it clear when incremental evaluation must propagate across segment boundaries.

- Interface attributes must be defined in pairs, a synthesized interface attribute and an inherited interface attribute. When a synthesized interface attribute in one segment changes in value, the corresponding inherited interface attribute in all segments must be reevaluated. This simplifies the dependencies among the interface attributes of segments.
- We limit the semantic equation for an inherited interface attribute of a segment to the **union** of the corresponding synthesized interface attributes of all segments. Thus, there is no need for explicit attributes and semantic equations for the root of the program.
- Segment linkage is performed by means of a special attribute of the distributable symbol, *system-module-name*. This attribute is defined as the concatenation of two attributes representing the name of the modular unit contained in the segment, and the name of the program that contains this modular unit. Both name attributes must be defined in each segment.

The architecture of the MERCURY editor generator is depicted in figure A-1. It is similar to the Synthesizer Generator on which it is based, except for the parts of the specification language that deal with segments (which were described above), and some of the kernel algorithms (which are discussed in section A.4 below).

A.3. Run-Time Support: The Attribute Propagation Layer

The run-time support, known as the attribute propagation layer (APL), is responsible for propagating changes in attribute information among the local editors. The APL is implemented as a separate process where the (virtual) root of the program resides. The root of the program is represented by a linked list of nodes, one for each segment, with each node containing a copy of the synthesized interface attributes of the segment. The APL is replicated on each workstation in the distributed development environment.

The APL can support editors for several programs simultaneously, even programs written in different languages. The current implementation, however, does not handle the transmission of changes between modules belonging to the same program that are written in different languages.

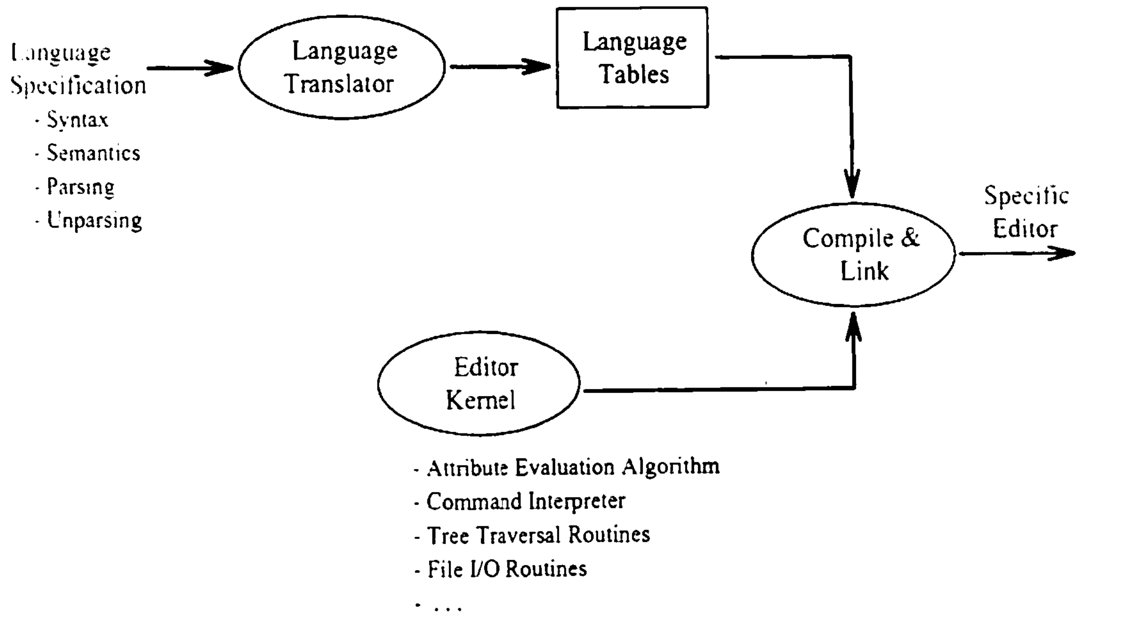


Figure A-1: *The MERCURY Editor Generator*

The control loop of the APL algorithm accepts messages from either (1) local editors running on the same machine, or (2) other APL processes running on other machines. Local editors can send the following types of messages to the APL:

- START-LOCAL-EDITOR — This message is sent when the local editor is first invoked. It sets up a communication channel between the local editor and the local APL process over which further communication takes place.

- INIT-MODULE — This message is sent when a new segment is created. If this is the first segment of a new program, a program node is created and added to the list of programs supported by the APL. A node corresponding to this new segment is then created, containing the segment's synthesized interface attributes, and added to the list of segments for the specified program.

  The APL then recomputes the value of each inherited interface attribute by concatenating the corresponding synthesized interface attribute from all segments, and sends the inherited interface attributes in SYSTEM-UPDATE messages to every local editor running on the same machine. Then, the APL forwards the INIT-MODULE message to APL processes running on other machines, so these can, in turn, update the segments that reside on those machines.

- MODULE_UPDATE — This message is sent when the value of a synthesized interface attribute of a segment changes as a result of a subtree replacement performed on the segment. The new value of the synthesized interface attribute replaces the one previously stored in the node for the segment in the APL. A SYSTEM-UPDATE message containing the new value of the corresponding inherited interface attribute is sent to all local editors, and the MODULE-UPDATE message is forwarded to other APLs.
- KILL-MODULE — This message is sent when a program segment is deleted. The APL deletes the segment node and all its associated synthesized interface attributes. Then it sends SYSTEM-UPDATE messages for each inherited interface attribute to the the local editors, and forwards the KILL-MODULE message to other APL.
- EXIT-EDITOR — This message is sent at the end of an editing session. If the segment was saved, then the APL sets the segment's status as ''dormant''. No messages are sent to dormant segments. If the segment was not saved, then the APL processes this message in the same way as a KILL-MODULE message.

As can be inferred from the description of how messages from the local editors are handled, an APL also receives INIT-MODULE, MODULE-UPDATE, and KILL-MODULE messages from other APL processes running on remote machines. These messages are handled in the same way as we described above, except that they are not forwarded to other APLs.

The APL contains additional algorithms for restoring consistency among the replicated data after machine or network failure; these algorithms, which require additional messages to be sent between APLs than those listed above, are described elsewhere [Kaiser 87]. Thus, programmers can continue working even when part of the network is broken or some machines are down, trusting that the editor will propagate changes affecting previously inaccessible modules as soon as the network or machine is brought back up.

A.4. Kernel Algorithms

The most interesting kernel algorithm is the distributed incremental attribute evaluation algorithm. This algorithm is a distributed version of the incremental attribute evaluation algorithm used in the Synthesizer Generator for single-user environments. The algorithm used in MERCURY differs in the following ways: (1) the program is split up into segments, each of which can be separately edited in a local editor, and (2) changes made to one segment can propagate into another segment, requiring the attribute evaluation algorithm within a local editor to handle multiple evaluations simultaneously.

A subtree replacement within a segment initiates an incremental evaluation process to reestablish consistency. In many cases, operation proceeds exactly as in the algorithm used in the Synthesizer Generator. But sometimes a synthesized interface attribute of an edited segment changes in value, and thus becomes inconsistent with the corresponding inherited interface attributes of the other segments. In this case, the new value of the synthesized interface attribute is transmitted to the APL in a MODULE-UPDATE message. The APL computes the new value of the corresponding inherited interface attribute and transmits it to all the segments in a SYSTEM-UPDATE message.

When an editor receives a SYSTEM-UPDATE message from the APL with a new value for an inherited interface attribute, it is treated as a simulated subtree replacement at the root of the segment; a new incremental evaluation process is initiated within the segment to reestablish consistency. Multiple evaluation processes initiated by the APL and by subtree replacements within a segment are merged to minimize costs when multiple processes may affect the same attribute, or cancel each other out, by the algorithms described in chapter 3.

Interface attributes are not propagated between a local editor and the APL when the value of the *system-module-name* attribute of the segment being edited is not completely defined (*i.e.*, either the program name, or the segment name, or both, have not yet been filled in by the programmer). A change in the value of this attribute from undefined to defined results in the transmission of an INIT-MODULE message to the APL, while changing this value from defined to undefined (*e.g.*, if the identifier defining the program name is deleted from the segment) results in the transmission of a KILL-MODULE message to the APL. MODULE-UPDATE messages contain information derived from the *system-module-name* attribute indicating the names of the program and the module to which the changed synthesized interface attribute belongs.

Several other algorithms of the Synthesizer Generator needed to be modified or created anew to handle multiple users and a distributed editing environment. Examples include the new editing commands to switch between the different granularities for propagating changes among segments, the protocol between the editor and the APL for initiating and terminating editing sessions, routines for displaying information about incoming and outgoing messages, and asynchronous I/O routines to read input from the APL or the keyboard.

Several programmers have expressed interest in an additional facility not currently included in MERCURY: *change simulation*. Change simulation would allow programmers to experiment with a module interface change, and see how it affects other modules, but without informing the other programmers responsible for those modules. The intent is to carry out ''what if'' experiments, as in spreadsheet systems [Ross 85], to determine the impact and extent of proposed changes.

We implemented a rudimentary version of change simulation, but then ran into the dilemma of how to present the results to the programmer who made the ''simulated'' change. In MERCURY, static semantic errors are normally indicated by displaying a string containing an error message next to the statement or expression containing the error. But this is problematic when this statement or expression was written by a different programmer within a different module, and the programmer who introduced the error through his module interface change has no understanding of the context of the statement or expression and how it interacts with the rest of the other programmer's module. It is clear that further user modeling research is needed to determine how best to handle this issue, so we have not incorporated the change simulation feature into MERCURY.

A.5. Conclusion

Implementing the MERCURY prototype proved to be an invaluable exercise because it helped identify the key problems involved in building a distributed editor based on attribute grammars. Elegant solutions to these key problems were presented in the thesis, but only some of these solutions were incorporated in the prototype due to time constraints.

The choice we made in implementing MERCURY by modifying the Synthesizer Generator made it abundantly clear why code reusability is (1) essential, and (2) extremely difficult. The Synthesizer Generator consists of over 50,000 lines of code. The only documentation we had access to was the sporadic comments in the code itself. Thus, even though both systems have a lot of common functionality, making even the smallest change required considerable time and effort. Having a tool like MERCURY to help analyze the impact of a change would have made the task somewhat easier!

Appendix B

Attribute Grammar for Distributed Calendar Application

Editor specifications for the MERCURY system are written in the Synthesizer Specification Language [Reps 89b], extended for segmentable attribute grammars as described in Appendix A.

The specifications for the calendar environment are divided into the following nine files:

1. **cal.x.ssl** — abstract and concrete syntax for appointments and meetings;
2. **cal.m.ssl** — semantics for appointments;
3. **cal.t.ssl** — attribute type definitions and semantic function definitions for appointments;
4. **cal.u.ssl** — unparsing specifications for appointments and meetings (*i.e.*, how information in each calendar is displayed to the user);
5. **cal2.m.ssl** — semantics for meetings;
6. **cal2.t.ssl** — attribute type definitions and semantic function definitions for meetings;
7. **cal.lexical.ssl** — lexical definitions of input tokens;
8. **cal.errors.ssl** — list of error messages;
9. **cal.transforms.ssl** — transformation declarations.

File 1: cal.x.ssl

```
/* Abstract syntax */

let format_strings = true;

root calendar;

calendar: Cal(group owner commands)
        ;

group: GroupBot()
        | GroupName(ID)
        ;

owner: OwnerBot()
        | Name(ID)
        ;

list commands;
commands: CommandsNil()
        | CommandsPair(command commands)
        ;

command: CommandBot()
        | MakeAppointment(month day from to note)
        | ScheduleMeeting(attendees duration month day month day purpose)
        | DisplayCalendar()
        | DisplayDay(month day)
        | DisplayPeriod(month day month day)
        ;

month: MonthBot()
        | TheMonth(INT)
        ;

day: DayBot()
        | TheDay(INT)
        ;

from: FromBot()
        | Beg(INT)
        ;

to: ToBot()
        | End(INT)
        ;

note: NoteBot()
        | For(STRING)
        ;

list attendees;
attendees: NoOne()
        | Two(person attendees)
        ;

person: PersonBot()
        | RealPerson(ID)
        ;

duration: TimeBot()
```

```
      | Time(INT)
      ;

purpose: PurposeBot()
      | ThePurpose(STRING)
      ;

/* Concrete syntax */

Calendar { syn calendar abs; };
Group { syn group abs; };
Owner { syn owner abs; };
Commands { syn commands abs; };
Command { syn command abs; };
Month { syn month abs; };
Day { syn day abs; };
From { syn from abs; };
To { syn to abs; };
Note { syn note abs; };
Attendees { syn attendees abs; };
Person { syn person abs; };
Duration { syn duration abs; };
Purpose { syn purpose abs; };

calendar     ~    Calendar.abs;
group        ~    Group.abs;
owner        ~    Owner.abs;
commands     ~    Commands.abs;
command      ~    Command.abs;
month        ~    Month.abs;
day          ~    Day.abs;
from         ~    From.abs;
to           ~    To.abs;
note         ~    Note.abs;
attendees    ~    Attendees.abs;
person       ~    Person.abs;
duration     ~    Duration.abs;
purpose      ~    Purpose.abs;

Calendar ::= (Group Owner Commands)
          { Calendar.abs = Cal(Group.abs, Owner.abs, Commands.abs); }
    ;

Group ::= (ID)
          { Group.abs = GroupName(ID); }
    ;

Owner ::= (ID)
          { Owner.abs = Name(ID); }
    ;

Commands ::= (Command)
          { Commands.abs = CommandsPair(Command.abs, CommandsNil()); }
    | (Command Commands)
          { Commands$1.abs = CommandsPair(Command.abs, Commands$2.abs); }
    ;

Command ::= ( Month '/' Day ',' From '-' To ':' Note )
          { Command.abs = MakeAppointment(Month.abs, Day.abs,
                    From.abs, To.abs, Note.abs); }
    | ( NEED '(' Attendees ')' FOR Duration SOMETIME BETWEEN Month
      '/' Day AND Month '/' Day RE Purpose )
```

```
            { Command.abs = ScheduleMeeting(Attendees.abs, Duration.abs,
                        Month$1.abs, Day$1.abs, Month$2.abs,
                        Day$2.abs, Purpose.abs); }
        | (DISPLAY ALL)
            { Command.abs = DisplayCalendar(); }
        | (DISPLAY DAY Month '/' Day)
            { Command.abs = DisplayDay(Month.abs, Day.abs); }
        | (DISPLAY PERIOD Month '/' Day TO Month '/' Day)
            { Command.abs = DisplayPeriod(Month$1.abs,
                  Day$1.abs, Month$2.abs, Day$2.abs); }
        ;

Month ::= (INTEGER)
            { Month.abs = TheMonth(STRtoINT(INTEGER)); }
        ;

Day ::= (INTEGER)
            { Day.abs = TheDay(STRtoINT(INTEGER)); }
        ;


From ::= (INTEGER)
            { From.abs = Beg(STRtoINT(INTEGER)); }
        ;

To ::= (INTEGER)
            { To.abs = End(STRtoINT(INTEGER)); }
        ;

Note ::=(STRING)
            { Note.abs = For(STRING); }
        ;

Attendees ::= (Person)
            { Attendees.abs = Two(Person.abs, NoOne()); }
        | (Person ',' Attendees)
            { Attendees$1.abs = Two(Person.abs, Attendees$2.abs); }
        ;

Person ::= (ID)
            { Person.abs = RealPerson(ID); }
        ;

Duration ::= (INTEGER)
            { Duration.abs = Time(STRtoINT(INTEGER)); }
        ;

Purpose ::= (STRING)
            { Purpose.abs = ThePurpose(STRING); }
        ;
```

File 2: cal.m.ssl

```
/* Semantics for appointments*/

/* Attribute declarations */

calendar {
        syn STR group_owner_name;
        interface syn SCHEDULE owners_appts;
        interface inh SCHEDULES all_appts;
        };
```

```
group {
      syn ID group_name;
      };

owner {
      syn ID name;
      };

commands, command {
      inh APPTS appts_in;
      syn APPTS appts_out;
      };

month, day {
      syn INT id;
      };

to, from {
      syn INT hr;
      };

note {
      syn STRING note;
      };

/* Semantic equations */

calendar: Cal {
        $$.group_owner_name = group.group_name # "%%" # owner.name;
        commands.appts_in = NullList();
        $$.owners_appts = Schedule(owner.name, commands.appts_out);
      }
      ;

group: GroupBot {
        $$.group_name = "?";
      }
      | GroupName {
        $$.group_name = ID;
      }
      ;

owner: OwnerBot {
        $$.name = "?";
      }
      | Name {
        $$.name = ID;
      }
      ;

commands: CommandsNil {
        $$.appts_out = $$.appts_in;
      }
      | CommandsPair {
        command.appts_in = $$.appts_in;
        commands$2.appts_in = command.appts_out;
        $$.appts_out = commands$2.appts_out;
      }
      ;

command: CommandBot {
        $$.appts_out = $$.appts_in;
```

```
}
| MakeAppointment {
  local ERR error1;
  local ERR error2;
  local ERR error3;
  local ERR error4;
  local ERR error5;

  error1 = is_date_valid(month.id, day.id);
  error2 = is_hour_valid(from.hr);
  error3 = is_hour_valid(to.hr);
  /* only check that the slot specified by from and to is */
  /* valid if the hours from and to are valid.          */
  error4 = ((error2 == NoErr) && (error3 == NoErr))
    ? is_time_slot_valid(from.hr, to.hr)
    : NoErr;
  /* only check that time slot is free if date and slot are valid */
  error5 = ((error1 == NoErr) && (error2 == NoErr) &&
        (error3 == NoErr) && (error4 == NoErr))
    ? is_free($$.appts_in, month.id, day.id, from.hr, to.hr)
      ? NoErr
      : Err1 /* <-- slot already filled */
    : NoErr;
  $$.appts_out = ((error1 == NoErr) && (error2 == NoErr) &&
             (error3 == NoErr) && (error4 == NoErr) &&
             (error5 == NoErr))
    ? insert(Appt(month.id, day.id, from.hr,to.hr, note.note),
          $$.appts_in)
    : $$.appts_in;
}
| ScheduleMeeting {
  $$.appts_out = $$.appts_in;
}
| DisplayCalendar {
  $$.appts_out = $$.appts_in;
}
| DisplayDay {
  local APPTS appts;
  local ERR error;

  $$.appts_out = $$.appts_in;
  error = is_date_valid(month.id, day.id);
  appts = (error == NoErr)
         ? find_appts_for_day_specified($$.appts_in,
                        month.id, day.id)
         : NullList();
}
| DisplayPeriod {
  local APPTS appts;
  local ERR error1;
  local ERR error2;
  local ERR error3;

  $$.appts_out = $$.appts_in;
  error1 = is_date_valid(month$1.id, day$1.id);
  error2 = is_date_valid(month$2.id, day$2.id);
  error3 = ((error1 == NoErr) && (error2 == NoErr))
         ? is_period_valid(month$1.id, day$1.id,
                   month$2.id, day$2.id)
         : NoErr;
  appts = ((error1 == NoErr) && (error2 == NoErr) &&
        (error3 == NoErr))
```

```
                ? find_appts_for_period_specified($$.appts_in,
                   month$1.id, day$1.id, month$2.id, day$2.id)
                : NullList();
      }
      ;

month: MonthBot { month.id = -1; }
      | TheMonth { month.id = INT; }
      ;

day: DayBot { day.id = -1; }
      | TheDay { day.id = INT; }
      ;

from: FromBot { from.hr = -1; }
      | Beg { from.hr = INT; }
      ;

to: ToBot { to.hr = -1; }
      | End { to.hr = INT; }
      ;

note: NoteBot { note.note = ""; }
      | For { note.note = STRING; }
      ;
```

File3: cal.t.ssl

```
/* Attribute type definitions for appointments */

list APPTS;
APPTS  : NullList() [ @ : ]
      | ListConcat( APPT APPTS ) [ @ : @ ["%n"] @ ]
      ;

        /* month day from  to  note  */
APPT   : Appt(INT  INT INT  INT STRING)
        [@ : "%t" @ "/" @ ", " @ " - " @ ": " @ "%b"]
      ;

list SCHEDULES;
SCHEDULES: NullSchedule() [ @ : ]
      | ScheduleConcat(SCHEDULE SCHEDULES) [ @ : @ ["%n"] @ ]
      ;

SCHEDULE: Schedule(ID APPTS) [ @ : @ ":%n" @ "%n" ]
      ;

/* function definitions */

/*
 * is_date_valid (mm, dd)
 *    Check if the date specified by mm/dd is valid.  Does not
 *    report error if either the month or the date are equal
 *    to the default value set by the bottom productions, -1.
 */

ERR is_date_valid(INT mm, INT dd)
{
  ((((mm < 1) || (mm > 12)) && (mm != -1))
    ? Err2 /* <-- month must be between 1 and 12 */
    : (((mm == 1) || (mm == 3) || (mm == 5) || (mm == 7) ||
        (mm == 8) || (mm == 10) || (mm == 12))
       && (((dd < 1) || (dd > 31)) && (dd != -1)))
       ? Err3 /* <-- day must be between 1 and 31 */
       : (((mm == 4) || (mm == 6) || (mm == 11))
         && (((dd < 1) || (dd > 30)) && (dd != -1)))
         ? Err4 /* <-- day must be between 1 and 30 */
         : ((mm == 2) && (((dd<1) || (dd>28)) && (dd != -1)))
           ? Err5 /* <-- day must be between 1 and 28 */
           : NoErr
  )
};

/*
 * is_hour_valid (hr)
 *    Checks that the hour, hr, is between 1 and 24 inclusive,
 *    or -1, the initial value.
 */

ERR is_hour_valid (INT hr)
{
  ((((hr < 1) || (hr > 24)) && (hr != -1))
    ? Err6 /* <-- hour must be between 1 and 24 */
    : NoErr)
};

/*
 * is_time_slot_valid (fr, to)
 *    Checks that the time slot specified by the hours "fr" and "to"
 *    is valid, that is, that "fr" is less than "to (we don't allow
 *    appts spanning  consecutive days).
 */

ERR is_time_slot_valid(INT fr, INT to)
{
  (((fr == -1) || (to == -1) || (fr < to))
    ? NoErr
    : Err7 ) /* <-- from must be less than to */
};

/*
 * overlap (s1, e1, s2, e2)
 *    Checks whether the time period s2 to e2, which is the time
 *    of the appointment being scheduled, overlaps with s1 to e1,
 *    the time of an appointment in schedule.
 */

BOOL overlap(INT s1, INT e1, INT s2, INT e2)
{
  /* no conflict if the second appointment ends before the first one */
  /* begins, or if the first appointment ends before the second one  */
  /* begins.                                                         */

  (((e2 <= s1) || (e1 <= s2))
    ? false
    : true)
};

/*
 * is_free (schedule, month, day, from, to)
 *    Returns true if the time slot specified is clear in the
 *    list of appointments, schedule.
 */
```

```
BOOL is_free(APPTS schedule, INT month, INT day, INT from, INT to)
{
  with (schedule) (
    NullList: true,
    ListConcat(head, tail):
      with (head) (
        Appt(m, d, f, t, *):
          ((m == month) && (d == day) && overlap(f,t,from,to))
            ? false
            : is_free(tail, month, day, from, to)
      )
  )
};


/*
 * insert (a, a_list)
 *    Inserts appt, a, in the right place in a list of appointments,
 *    a_list, where the list is ordered by the date and time of the
 *    appointment.  Note that a does not overlap with any entries
 *    in a_list.
 */

APPTS insert(APPT a,  APPTS a_list)
{
  with(a_list) (
    NullList: a :: NullList(),
    ListConcat(head,tail):
      with(head) (
        Appt(m, d, f, t, *):
          with(a) (
            Appt(a_m, a_d, a_f, a_t, *):
              ((a_m > m) ||
               ((a_m == m) && (a_d > d)) ||
               ((a_m == m) && (a_d == d) && (a_f >= t)))
                 ? head :: insert(a, tail)
                 : a :: a_list
          )
      )
  )
};


/*
 * find_appts_for_day_specified ( a_list, mm, dd )
 *    Selects from a_list, an ordered list of appointments, those
 *    appointments for the day specified by the month, mm and the
 *    day, dd.
 */

APPTS find_appts_for_day_specified ( APPTS a_list, INT mm, INT dd )
{
  with(a_list) (
     NullList: NullList(),
     ListConcat(head,tail):
       with (head) (
         Appt(m, d, f, t, *):
           ((m == mm) && (d == dd))
             ? head :: find_appts_for_day_specified(tail, mm, dd)
             : ((m < mm) || ((m == mm) && (d < dd)))
               ? find_appts_for_day_specified(tail, mm, dd)
               : NullList()
        )
    )
  };

/*
 * find_appts_for_period_specified (a_list, mm1, dd1, mm2, dd2)
 *     Selects from a_list, an ordered list of appointments, those
 *     appointments for the period specified.
 */

APPTS find_appts_for_period_specified ( APPTS a_list, INT mm1, INT dd1,
                                      INT mm2, INT dd2  )
{
  with(a_list) (
     NullList: NullList(),
     ListConcat(head,tail):
       with (head) (
         Appt(m, d, f, t, *):
           is_in_period(m, d, mm1, dd1, mm2, dd2)
             ? head ::
               find_appts_for_period_specified(tail, mm1, dd1, mm2, dd2)
             : ((m < mm1) || ((m == mm1) && (d < dd1)))
               ? find_appts_for_period_specified(tail, mm1, dd1,
                                         mm2, dd2)
               : NullList()
       )
  )
};


/*
 * is_in_period (m, d, mm1, dd1, mm2, dd2)
 *     Returns true if the day m/d falls within the period,
 *     mm1/dd1 to mm2/dd2 inclusive.
 */

BOOL is_in_period( INT m, INT d, INT mm1, INT dd1, INT mm2, INT dd2)
{
  (((m > mm1) && (m < mm2)) ||
   ((m == mm1) && (m == mm2) && (d >= dd1) && (d <= dd2)) ||
   ((m == mm1) && (m != mm2) && (d >= dd1)) ||
   ((m != mm1) && (m == mm2) && (d <= dd2)))
};

/*
 * is_period_valid (mm1, dd1, mm2, dd2)
 *     Returns true if the date mm1/dd1 comes before mm2/dd2.
 *     The two dates are guaranteed to be valid.
 */

ERR is_period_valid ( INT mm1, INT dd1, INT mm2, INT dd2)
{
  ((((mm1 == -1) || (dd1 == -1) || (mm2 == -1) || (dd2 == -1)) ||
    (mm1 < mm2) ||
    ((mm1 == mm2) && (dd1 < dd2)))
      ? NoErr
      : Err8()) /* <-- first date must be less than the second */
};
```

File 4: cal.u.ssl

```
/* Unparsing */
```

```
calendar: Cal[ @ ::= "                    Group  " @ "%n%n"
            "              Appointment Manager for " @ "%n%n"
            s1 calendar.pending_mtg_requests s2 @ "%n%n"]
     ;

group: GroupBot[ ^ ::= "<group>" ]
     | GroupName[^ ::= ^ ]
     ;

owner: OwnerBot[ ^ ::= "<owner>" ]
     | Name[^ ::= ^ ]
     ;

commands: CommandsNil[^ : ]
     | CommandsPair[^ : ^  ["%n%n"] @]
     ;

command: CommandBot[^ ::= "<command>"]
     | MakeAppointment[@ ::= @ "/" @ error1 ", " @ error2 " - " @
                         error3 error4 error5 ": " @]
     | ScheduleMeeting[@ ::= "schedule (" @ ") for " @ " hrs " error1
                    " sometime between "
                    @  "/" @ error2 " and " @ "/" @ error3 error4
                    " re: " @ "%t%t%n" error date "%n%b%b"]
     | DisplayCalendar[@ ::= "%t%nAppointments:%t%n"
                    command.appts_out "%b%b"]
     | DisplayDay[@ ::= "display calendar for " @ "/" @ error "%n%t%n"
                "Appointments, " month.id "/" day.id
                ":%t%n" appts "%b%b"]
     | DisplayPeriod[@ ::= "display calendar from " @ "/" @ error1
                " to " @ "/" @ error2 error3  "%n%t%n"
                "Appointments, " month$1.id "/" day$1.id " to "
                month$2.id "/" day$2.id
                ":%t%n" appts "%b%b"]
     ;

month: MonthBot[^ ::= "<month>"]
     | TheMonth[^ ::= ^]
     ;

day: DayBot[^ ::= "<day>"]
     | TheDay[^ ::= ^]
     ;

from: FromBot[^ ::= "<from>"]
     | Beg[^ ::= ^]
     ;

to: ToBot[^ ::= "<to>"]
     | End[^ ::= ^]
     ;

note: NoteBot[^ ::= "<note>"]
     | For[^ ::= ^]
     ;

attendees: NoOne[^ : ]
     | Two[^ : ^ [", "] @]
     ;

person: PersonBot[ ^ ::= "<name>" ]
     | RealPerson[ ^ ::= ^ error ]
```

```
      ;

duration: TimeBot[ ^ ::= "<time>" ]
      | Time[ ^ ::= ^ ]
      ;

purpose: PurposeBot[^ ::= "<purpose>"]
      | ThePurpose[ ^ ::= ^]
      ;
```

File5: cal2.m.ssl

```
/* Semantics for meetings */

/* Attribute declarations */

calendar {
      interface syn ALL_MTGS_ITEM owners_mtgs;
      interface inh ALL_MTGS_LIST all_mtgs_requested;
      syn MTGS pending_mtg_requests;
      };

commands, command {
      inh MTGS mtgs_in;
      syn MTGS mtgs_out;
      };

attendees, person {
      inh ATTENDEES att_in;
      syn ATTENDEES att_out;
      };

purpose {
      syn STRING note;
      };

duration {
      syn INT hrs;
      };

/* Semantic equations */

calendar: Cal {
         local STR s1;
         local STR s2;

         commands.mtgs_in = NullMtgList();
         $$.owners_mtgs = MtgsItem(owner.name, commands.mtgs_out);
         $$.pending_mtg_requests =
            f($$.all_mtgs_requested, $$.all_appts);
         s1 = ($$.pending_mtg_requests == NullMtgList)
            ? ""
            : "%t%nPending Meeting Requests:%t%n";
         s2 = ($$.pending_mtg_requests == NullMtgList)
            ? ""
            : "%b%b%n%n";
      }
      ;

commands: CommandsNil {
         $$.mtgs_out = $$.mtgs_in;
      }
```

```
        | CommandsPair {
          command.mtgs_in = $$.mtgs_in;
          commands$2.mtgs_in = command.mtgs_out;
          $$.mtgs_out = commands$2.mtgs_out;
        }
        ;

command: CommandBot {
          $$.mtgs_out = $$.mtgs_in;
        }
        | MakeAppointment {
          $$.mtgs_out = $$.mtgs_in;
        }
        | DisplayCalendar {
          $$.mtgs_out = $$.mtgs_in;
        }
        | DisplayDay {
          $$.mtgs_out = $$.mtgs_in;
        }
        | DisplayPeriod {
          $$.mtgs_out = $$.mtgs_in;
        }
        | ScheduleMeeting {
          local ERR error1;
          local ERR error2;
          local ERR error3;
          local ERR error4;
          local BOOL ready_to_schedule;
          local CHUNK_LIST possible_days;
          local CHUNK_LIST possible_times;
          local WHEN mtg_time;
          local ERR error;
          local STR date;

          attendees.att_in = NoAtt();
          error1 = is_duration_valid(duration.hrs);
          error2 = is_date_valid(month$1.id, day$1.id);
          error3 = is_date_valid(month$2.id, day$2.id);
          error4 = ((error2 == NoErr) && (error3 == NoErr))
                    ? is_meeting_period_valid(month$1.id, day$1.id,
                                month$2.id, day$2.id)
                    : NoErr;
          ready_to_schedule = ((error1 == NoErr) && (error2 == NoErr) &&
                         (error3 == NoErr) && (error4 == NoErr) &&
                         (month$1.id != -1) && (day$1.id != -1) &&
                         (month$2.id != -1) && (day$2.id != -1));
          possible_days = ready_to_schedule
                    ? chunk(month$1.id, day$1.id, 9,
                         month$2.id, day$2.id, 18)
                    : NoChunks();
          possible_times = ready_to_schedule
                    ? break_chunks(possible_days, duration.hrs)
                    : NoChunks();
          mtg_time = ready_to_schedule
                    ? schedule(possible_times, attendees.att_out,
                         {calendar.all_appts}, purpose.note)
                    : When(-1, -1, -1, -1);
          /* When(-9, -9, -9, -9): meeting could not be scheduled */
          /*              with given constraints.      */
          /* When(-1, -1, -1, -1): meeting not scheduled because */
          /*              of erroneous/missing dates.  */
          date = ((mtg_time == When(-9, -9, -9, -9)) ||
            (mtg_time == When(-1, -1, -1, -1)))
                ? ""
                : "meeting scheduled for " # date_to_str(mtg_time);
        error = (mtg_time == When(-9, -9, -9, -9))
                ? Err10 /* <-- cannot schedule this meeting */
                : NoErr;
        $$.mtgs_out =
            ((error == NoErr) && (mtg_time != When(-1, -1, -1, -1)))
                ? Mtg(attendees.att_out, mtg_time, purpose.note) ::
                  $$.mtgs_in
                : $$.mtgs_in;
    }
    ;

attendees: NoOne {
        $$.att_out = $$.att_in;
    }
    | Two {
        person.att_in = $$.att_in;
        attendees$2.att_in = person.att_out;
        $$.att_out = attendees$2.att_out;
    }
    ;

person: PersonBot {
        $$.att_out = $$.att_in;
    }
    | RealPerson {
        local ERR error;

        error = is_group_member(ID, {calendar.all_appts});
        $$.att_out = (error == NoErr)
            ? ($$.att_in @ (ID :: NoAtt())) /* so list is not */
                                            /* reversed.     */
            : $$.att_in;
    }
    ;

duration: TimeBot {
        $$.hrs = -1;
    }
    | Time {
        $$.hrs = INT;
    }
    ;


purpose: PurposeBot { purpose.note = ""; }
    | ThePurpose{ purpose.note = STRING; }
    ;
```

File 6: cal2.t.ssl

```
/* Attribute type definitions for meetings */

list ALL_MTGS_LIST;
ALL_MTGS_LIST : EmptyMtgsList() [ @ : ]
    | MtgsListConcat( ALL_MTGS_ITEM ALL_MTGS_LIST ) [ @ : @ ["%n"] @ ]
    ;

ALL_MTGS_ITEM: MtgsItem( ID  MTGS ) [ @ : @ ":%n" @ "%n" ]
    ;
```

```
list MTGS;
MTGS   : NullMtgList() [ @ : ]
     | MtgListConcat( MTG MTGS ) [ @ : @ ["%n"] @ ]
     ;


           /* date/time purpose */
MTG    : Mtg( ATTENDEES WHEN     STRING )
       [@ : "%t" "(" @ "), " @ ": " @ "%b"]
     ;

list ATTENDEES;
ATTENDEES: NoAtt() [ @ : ]
     | AttConcat( ID ATTENDEES ) [ @ : @ [", "] @ ]
     ;

     /* month day from to */
WHEN: When( INT   INT INT  INT ) [ @ : @ "/" @ ", " @ " - " @ ]
     ;

      /* month day from(>=9) to(<=18) */
CHUNK: Chunk( INT  INT INT      INT ) [ @ : @ "/" @ ", " @ " - " @ ]
     ;

list CHUNK_LIST;
CHUNK_LIST: NoChunks() [ @ : ]
     | ChunkConcat( CHUNK CHUNK_LIST ) [ @ : @ ["%n"] @ ]
     ;

/* function definitions */

/*
 * is_duration_valid (hrs)
 *    Checks if the duration of a meeting, hrs, is less than or
 *    equal to 9.  Meetings are only scheduled between 9am and
 *    6 pm.
 */

ERR is_duration_valid(INT hrs)
{
  ((hrs == -1) || ((hrs > 0) && (hrs <= 9))
    ? NoErr
    : Err11) /* <-- meeting length must be between 1 and 9 hours */
};

/*
 * is_meeting_period_valid (mm1, dd1, mm2, dd2)
 *    Returns true if the date mm1/dd1 is equal to, or comes
 *    before mm2/dd2.  The two dates are guaranteed to be valid.
 */

ERR is_meeting_period_valid ( INT mm1, INT dd1, INT mm2, INT dd2)
{
  ((((mm1 == -1) || (dd1 == -1) || (mm2 == -1) || (dd2 == -1)) ||
   (mm1 < mm2) ||
   ((mm1 == mm2) && (dd1 <= dd2)))
    ? NoErr
    : Err12) /* <-- first date must be less than or equal to the second */
};

/*
 * is_group_member (name, all_appts)
 *    Returns true if the person, name, has a calendar, in which
 *    case he would have an entry in all_appts.
 */

ERR is_group_member(ID name, SCHEDULES all_appts)
{
  with(all_appts) (
    NullSchedule: Err9, /* <-- person not member of group */
    ScheduleConcat(head, tail):
      with (head) (
        Schedule(n, *): (n == name) ? NoErr : is_group_member(name, tail)
      )
  )
};

/*
 * lookup_appts (all_appts, name)
 *    Returns the list of appointments for the specified person.
 *    We only call this function with a name that has been put in
 *    all_appts, so it is
 *    not possible for it not to be found; however, we include
 *    the case for NullSchedule() because it is required by SSL.
 */

APPTS lookup_appts ( SCHEDULES all_appts, ID name)
{
  with (all_appts) (
    NullSchedule: NullList(),
    ScheduleConcat(head, tail):
      with (head) (
        Schedule (id, appts):
          (id == name) ? appts : lookup_appts(tail, name)
      )
  )
};


/*
 * chunk (mm1, dd1, tt1, mm2, dd2, tt2)
 *    Creates a list of chunks of free time between mm1/dd1/tt1 and
 *    mm2/dd2/tt2.  tt1 and tt2 are guaranteed to be between
 *    9 and 18 inclusive, which is the time during which
 *    meetings can be scheduled.  The biggest chunk is
 *    all day, which is the chunk m/d/9/18.
 */

CHUNK_LIST chunk (INT mm1, INT dd1, INT tt1, INT mm2, INT dd2, INT tt2)
{
  (((mm1 == mm2) && (dd1 == dd2))
    ? Chunk(mm1, dd1, tt1, tt2) :: NoChunks()
    : ((mm1 == mm2) && (dd1 < dd2))
      ? (chunk_days(mm1, dd1, tt1, dd2-1, 18) @
        chunk(mm1, dd2, 9, mm1, dd2, tt2))
      : (mm1 < mm2)
        ? (chunk_mos(mm1, mm2-1) @
          (chunk_days(mm2, 1, 9, dd2-1, 18) @
           chunk(mm2, dd2, 9, mm2, dd2, tt2)))
        : NoChunks() /* never true */
  )
};

/*
 * chunk_days (m, dd1, tt1, dd2, tt2)
```

```
  )
};

/*
 * schedule (possible_times, p, all_appts, purpose) &
 * ok_with_rest (p, all_appts, m, d, f, t, s)
 *
 *    Finds a time for scheduling a meeting, where possible_times
 *    contains a list of chunks, where each chunk is exactly the
 *    length of the meeting; p is the list of attendees; all_appts
 *    contains everybody's calendar; and purpose describes
 *    what the meeting is for.  The list of possible_times is
 *    examined until either a meeting time is found that does
 *    not conflict with any attendee's schedule, or the list
 *    of possible times is exhausted, in which case the meeting
 *    cannot be scheduled.
 *
 *    The way to agree to a meeting is to enter an appointment
 *    in your calendar with the exact information as the meeting.
 *    Therefore, we check that if there is an appointment with
 *    the same values as the meeting being scheduled, that this
 *    does not cause a conflict.
 */

WHEN schedule ( CHUNK_LIST possible_times, ATTENDEES p, SCHEDULES all_appts,
          STRING purpose)
{
  with(possible_times) (
    NoChunks: When(-9, -9, -9, -9), /* can't schedule */
    ChunkConcat(head, tail):
      with(head) (
        Chunk(m, d, f, t):
          (ok_with_rest(p, all_appts, m, d, f, t, purpose))
            ? When(m, d, f, t)
            : schedule(tail, p, all_appts, purpose)
      )
  )
};

BOOL ok_with_rest(ATTENDEES p, SCHEDULES all_appts, INT m, INT d,
           INT f, INT t, STRING s )
{
  with(p) (
    NoAtt(): true,
    AttConcat(head,tail):
      (is_free(lookup_appts(all_appts, head), m, d, f, t) ||
       conflict_with_appt_for_same_meeting(lookup_appts(all_appts, head),
                               m, d, f, t, s))
        ? ok_with_rest(tail, all_appts, m, d, f, t, s)
        : false
  )
};

BOOL conflict_with_appt_for_same_meeting (APPTS a, INT mo, INT da,
                        INT fr, INT to, STRING str)
{
  with (a) (
    NullList: false,
    ListConcat(head, tail):
      with (head) (
        Appt(m, d, f, t, s):
          ((m == mo) && (d == da) && (f == fr) &&
 *    Returns free chunks for the days dd1/tt1 to dd2/tt2 of
 *    the same month, m.
 */

CHUNK_LIST chunk_days ( INT m, INT dd1, INT tt1, INT dd2, INT tt2 )
{
  ((dd1 == dd2)
    ? Chunk(m, dd1, tt1, tt2) :: NoChunks()
    : Chunk(m, dd1, tt1, 18) :: chunk_days(m, dd1+1, 9, dd2, tt2))
};

/*
 * chunk_mos (mm1, mm2)
 *    Returns free chunks for the period of time between
 *    mm1/1 to mm2/30.  I'm  taking a shortcut here and considering
 *    all months to have 30 days.
 */

CHUNK_LIST chunk_mos ( INT mm1, INT mm2 )
{

  ((mm1 == mm2)
    ? chunk_days(mm1, 1, 9, 30, 18)
    : chunk_days(mm1, 1, 9, 30, 18) @ chunk_mos(mm1+1, mm2))
};

/*
 * break_chunks (free_chunks, length) & small_chunks (ch, length)
 *    Breaks up each chunk in the list of free chunks, free_chunks,
 *    even smaller chunks, where each small chunk is exactly
 *    length hours.
 */

CHUNK_LIST break_chunks( CHUNK_LIST free_chunks, INT length)
{
  with (free_chunks) (
    NoChunks: NoChunks(),
    ChunkConcat(head,tail):
      with(head) (
        Chunk(m, d, f, t):
          ((t-f) < length) /* no good -- throw away chunk */
            ? break_chunks(tail, length)
            : ((t-f) == length)
              ? (head :: break_chunks(tail, length))
              : /* (t-f) > length */
               (small_chunks(head, length) @
                break_chunks(tail, length))
      )
  )
};

CHUNK_LIST small_chunks ( CHUNK ch, INT length)
{
  with (ch) (
    Chunk(m, d, f, t):
      ((t-f) < length)
        ? NoChunks()
        : ((t-f) == length )
          ? (Chunk(m, d, f, t) :: NoChunks())
          : /* (t-f) > length */
           (Chunk(m, d, f, f+length) ::
            small_chunks(Chunk(m, d, f+1, t), length))
          (t == to) && (s == str))
           ? true
           : conflict_with_appt_for_same_meeting(tail,
                mo, da, fr, to, str)
      )
  )
};

/*
 * date_to_str (date)
 *     Converts date, which is made of four integers representing
 *     the month, the day, from and to respectively, to
 *     a string.
 */

STR date_to_str (WHEN date)
{
  with (date) (
    When(m, d, f, t):
      INTtoSTR(m) # "/" # INTtoSTR(d) # ", " #
         INTtoSTR(f) # " - " # INTtoSTR(t)
  )
};

/*
 * f (all_mtgs_requested, all_appts) &
 * g (all_mtgs_requested, all_appts)
 *
 *     Returns the list of meetings that have been requested but
 *     which have not yet been agreed on by all of the attendees.
 */

MTGS f (ALL_MTGS_LIST all_mtgs_requested, SCHEDULES all_appts)
{
  with (all_mtgs_requested) (
    EmptyMtgsList: NullMtgList(),
    MtgsListConcat(head,tail):
      with(head) (
        MtgsItem(*, mtgs): g(mtgs, all_appts) @ f(tail, all_appts)
      )
  )
};

MTGS g ( MTGS m, SCHEDULES all_appts)
{
  with (m) (
    NullMtgList: NullMtgList(),
    MtgListConcat(h, t):
      with (h) (
        Mtg(people, date, purpose):
          is_mtg_accepted(people, date, purpose, all_appts)
            ? g(t, all_appts)
            : h :: g(t, all_appts)
      )
  )
};

/*
 * is_mtg_accepted (people, date, purpose)
 *     Returns true if all the attendees have accepted the
 *     meeting for the date and purpose specified.  A person
 *     accepts a meeting by entering it as an appointment in
 *    his calendar, with exact matches for the date and
 *    purpose required.
 */

BOOL is_mtg_accepted (ATTENDEES people, WHEN date, STRING purpose,
              SCHEDULES all_appts)
{
  with (people) (
    NoAtt: true,
    AttConcat(head, tail):
      with (date) (
        When(m, d, f, t):
          has_entry(lookup_appts(all_appts, head), m, d, f, t, purpose)
            ? is_mtg_accepted(tail, date, purpose, all_appts)
            : false
      )
  )
};

/*
 * has_entry (appts, mo, da, fr, to, purpose)
 *    Returns true if there is an entry in appts for the meeting
 *    on the date, mo/da, between the hours, fr and to, for the
 *    purpose specified.
 */

BOOL has_entry (APPTS appts, INT mo, INT da, INT fr, INT to, STRING purpose)
{
  with (appts) (
    NullList: false,
    ListConcat(head, tail):
      with (head) (
        Appt(m, d, f, t, for):
          ((m == mo) && (d == da) && (f == fr) && (t == to) &&
           (for == purpose))
            ? true
            : has_entry(tail, mo, da, fr, to, purpose)
      )
  )
};
```

File 7: cal.lexical.ssl

```
/* Lexical syntax */

NEED: < "need" >;
FOR: < "for" >;
RE: < "re" >;
DISPLAY: < "display" >;
DAY: < "day" >;
PERIOD: < "period" >;
TO: < "to" >;
ALL: < "all" >;
SOMETIME: < "sometime" >;
BETWEEN: < "between" >;
AND: < "and" >;
INTEGER: < [0-9]+ >;
WHITESPACE: < [\ \t\n] >;
ID: < [a-zA-Z][a-zA-Z0-9]*|[?] >;
STRING: < ([^\n]+) >;
```

File 8: cal.errors.ssl

```
/* Errors and their print representation */

ERR:    NoErr()[ @ : "" ]
     | Err1() [@: " { <-- slot already filled } " ]
     | Err2() [@: " { <-- month must be between 1 and 12 } " ]
     | Err3() [@: " { <-- day must be between 1 and 31 } " ]
     | Err4() [@: " { <-- day must be between 1 and 30 } " ]
     | Err5() [@: " { <-- day must be between 1 and 28 } " ]
     | Err6() [@: " { <-- hour must be between 1 and 24 } " ]
     | Err7() [@: " { <-- from must be less than to } " ]
     | Err8() [@: " { <-- first date must be less than the second } " ]
     | Err9() [@: " { <-- person not member of group } " ]
     | Err10() [@: " { <-- cannot schedule this meeting } " ]
     | Err11() [@: " { <-- meeting length must be between 1 and 9 } " ]
     | Err12() [@: " { <-- first date must be less than or equal to the second } " ]
     ;
```

File 9: cal.transforms.ssl

```
/* Template commands */

transform calendar
     on "calendar"
          <calendar>: Cal(<group>, <owner>, <commands>) ;

transform command
     on "make-appointment"
          <command>: MakeAppointment(<month>, <day>, <from>,<to>, <note>),
     on "schedule-meeting"
          <command>: ScheduleMeeting(<attendees>, <duration>,
                <month>, <day>, <month>, <day>, <purpose>),

     on "display-calendar"
          <command>: DisplayCalendar(),

     on "display-day"
          <command>: DisplayDay(<month>, <day>),

     on "display-period"
          <command>: DisplayPeriod(<month>, <day>, <month>, <day>)
     ;
```