DYMOS: A DYNAMIC MODIFICATION SYSTEM

by

INSUP LEE

Computer Sciences Technical Report #503

May 1983

DYMOS: A DYNAMIC MODIFICATION SYSTEM

by

INSUP LEE

A thesis submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1983

DYMOS: A DYNAMIC MODIFICATION SYSTEM

Insup Lee

Under the supervision of Assistant Professor Robert P. Cook

DYMOS supports a single programmer modifying a module-based program dynamically (that is, without stopping its execution). In DYMOS, the programmer modifies and recompiles the source code of procedures and modules that need to be replaced. The programmer then requests the system to change the current core image to incorporate new code and data. New object code is inserted by a dynamic modification process that is executed in parallel with other user processes.

Traditionally, modifications to running programs have been done by patching machine code. Our approach has several advantages over traditional machine code patching. For example, the current source code always represents the machine code in execution. In particular, only changes that will leave the program compile-time correct are allowed. Furthermore, the programmer need not know the details of the machine code generated by a compiler. Finally, it is easier to determine what part of the program to modify.

We first describe DYMOS. We then discuss how changes to a running program can be carried out in incremental steps. Finally, we propose an architecture that supports the synchronization and mutual exclusion capabilities necessary for dynamic modification.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Robert P. Cook, for his constant encouragement, limitless patience, and sound advice in this work, and for his diligence in reading and improving the early versions of this dissertation.

I would also like thank to the other members of my committee: Charles Fischer, James Goodman, Anthony Klug, and Charles Kime. In particular, I would like to thank Charles Fischer, who taught me much of what I know about programming languages and compiler construction and who provided many valuable suggestions in this work, and James Goodman for proposing improvements to the organization of this thesis.

I wish to thank my friends Tae Ho Bark, Sinsup Cho, Jin Keon Pai, and Paul Robinson for making my stay in Madison pleasant and memorable, and the Milwaukee Brewers and UW hockey team whose good records in past years made my stay in Madison enjoyable.

Finally, and most importantly, I want to thank my parents, Dr. and Mrs. Choo Hyung Lee, and my wife, Kisung, for their endless support and never-fading faith in me.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Introduction and Motivation

Programs are frequently modified during their lifetime to fix bugs, to make improvements, to add new capabilities, to delete obsolete features, or to adjust to new environments. Furthermore, modifications to some systems have to be made on the fly; that is, without stopping their execution. For example, changes to air traffic control systems, airline reservation systems, airborne programs, office systems, and telephone switching systems have to be performed dynamically. Other continuously running systems, such as operating systems, may be modified on the fly to increase their availability.

Traditionally, modifications to running programs have been done by patching machine code. However, patching is generally acknowledged to be a bad programming practice since it is highly error-prone. Glass [22] summarizes the difficulties with patching as follows:

(1) Patches are coded in a numeric and specialized language with which the programmer may be unfamiliar.

(2) Patches must be assigned vacant storage in memory. This task is non-trivial; for example, placing a patch to to an already assigned patch area is a common problem.

(3) Patches are only a temporary expedient. The real correction must be made in program source code. That is, the correction is done twice, probably using different algorithms.

(4) Patches are inserted into the computer using unusual techniques. For example, patches may be entered from its console, which is an error-prone process in itself and leaves no written record. Alternatively, patches may be punched into "binary" card decks and then read into the computer. Both processes are error-prone since no equipment has ever been defined for punching binary card decks and since such card decks can only be read into the computer by

disabling a card-reader's check-sum error-check.

Another traditional approach is to use duplicated systems so that one system can be modified while the other runs. For example, telephone switching systems are often implemented on two computers to provide continuous service [4]. Although a duplicated system can provide more reliable service while it is changed dynamically, it has the obvious disadvantage of requiring twice the resource needed for a non-duplicated system. Furthermore, the structure of the system becomes more complex since the system has to be designed to allow control to be switched from one system to the other.

This dissertation describes an integrated programming system that supports changes to a running program. In our system, the source code of selected procedures and modules is modified and recompiled. Then, the new object code is merged into the running program by the run-time support system. Four advantages to the programmer result from dealing with the source program instead of machine code.

(1) It is easier to determine what needs to be dynamically modified since programs written in high-level languages are easier to understand than the machine code generated by a compiler.

(2) The programmer does not have to know the machine code or the sequence of code generated by a compiler. Furthermore, the storage allocation and deallocation problem is handled by the system. Therefore, modification process is free from human errors that might result from misunderstanding of machine-dependent details.

(3) The current source program really represents the machine code in execution. This condition is necessary since the source program often serves as the most accurate description of a system. "Depatching" into the corresponding source code change is non-trivial, especially if high-level languages are used [32].

(4) The correctness of modification can be established more easily when only syntactically and semantically correct modifications are allowed.

1.2. Scope and Goals

The theme of this dissertation is that it is feasible to build an integrated programming system that supports the dynamic modification of a program, written in a concurrent high-level programming language. The integrated programming system consists of a command interpreter, editor, source code manager, compiler, and run-time support system. Although our system is developed based on a particular programming language, namely StarMod [8], the system can support different programming languages if their compilers are modified to generate the symbolic information and object code described in Chapter 6. However, we have not attempted to make our system language-independent.

The central issue of our research is the identification and justification of the necessary dynamic modification capabilities. Based on our implementation and use experience with a prototype dynamic modification system, we propose extensions to the programming language and a set of user commands. We also investigate the implementation issues of the system. In particular, how to manage source code and how to compile the new version of procedures and modules are addressed. Furthermore, we examine a run-time program representation that supports dynamic modification.

A running program might behave unexpectedly if it is changed at an arbitrary moment. Thus, we investigate a way to specify when changes to the running program are to be carried out. We also develop a methodology that can be used to find such a specification. Some changes to a program require many procedures and modules be modified. However, replacing many procedures and modules simultaneously might be unacceptable for some programs. We investigate how a running program can be modified incrementally.

1.3. Example Session

As an illustration of how our system functions, let us consider the simple on-line banking system in Figure 1-1. This on-line banking system was written specifically to illustrate the use of our system and is used in examples throughout the thesis whenever possible. (The complete version of the simple on-line banking system is in Appendix B.) This system receives a Deposit, Withdraw, Open, or Print request from the user's terminal and takes an appropriate action. The system is implemented using five modules: BankingSystem, BookKeeper, NameStorage, RequestHandler, and InputOutput.

The BookKeeper module provides routines to manage available account numbers and to fetch and to change information associated with each account (see Figure 1-2). It uses the NameStorage module to store customer names, where each name is stored in an array of 40 characters.

The RequestHandler module provides routines to handle customer's requests (see Figure 1-3). The InputOutput module of the RequestHandler module provides routines to read requests and to write requested information to the user's terminal. The ProcessRequest procedure continuously reads one request at a time and takes an appropriate action. The format of a request is as follows:

```
<request> ::= Deposit <account number> <amount> |
              Withdraw <account number> <amount> |
              Open <name> |
              Print <account number>
```

An <account number> is an integer between 100 and 200. A <name> can contain up to 80 characters; however, only the first 40 characters are stored. An <amount> is a positive integer for deposit and a negative integer for withdraw. For Deposit and Withdraw requests, the balance of an account is adjusted by a given amount. For an Open request, a new account number is assigned to a name. For a Print request, the name and balance of an account are printed.

```
module BankingSystem;
  const minAccountNo = 100; maxAccountNo = 200; maxStringSize = 80;
  type string = array 1 : maxStringSize of char;

  module BookKeeper;
    export StoreName, GetName, GetNewAccount, AdjustBalance, GetBalance;
    import minAccountNo, maxAccountNo, string;
    module NameStorage;
      export ChangeIntoName, ChangeIntoString, nameType;
      import string;
      procedure ChangeIntoName (var name: nameType; name: string);
      procedure ChangeIntoString (name: nameType; var str: string);
    end NameStorage;
    procedure StoreName (acnt: integer; str: string);
    procedure GetName (acnt: integer; var str: string);
    procedure GetNewAccount : integer;
    procedure AdjustBalance (acnt, amt : integer);
    procedure GetBalance (acnt : integer) : integer;
  end BookKeeper;

  module RequestHandler;
    export ProcessRequest;
    import string, GetName, StoreName,
         GetNewAccount, AdjustBalance, GetBalance;
    module InputOutput;
      export ReadTransType, ReadName, PrintName,
           ReadAccountNo, PrintAccountNo, ReadAmount, PrintBlance;
      import string, transType;
      procedure ReadTransType (var trans: transType);
      procedure ReadName (var name : string);
      procedure PrintName (name : string);
      procedure ReadAccountNo (var acnt : integer);
      procedure PrintAccountNo (acnt : integer);
      procedure ReadAmount (var amt : integer );
      procedure PrintBlance (amt : integer );
    end InputOutput;
    procedure PrintAccount;
    procedure OpenAccount;
    procedure ProcessRequest;
  end RequestHandler;

begin
  ProcessRequest;
end BankingSystem;
```

Figure 1-1. Outline of the simple on-line banking system.

```
module BookKeeper;
  export StoreName, GetName, GetNewAccount, AdjustBalance, GetBalance;
  import minAccountNo, maxAccountNo, string;

  module NameStorage;
    export StoreName, GetName, nameType;
    import string;
    const nameSize = 40;
    type nameType = array 1 : nameSize of char;

    procedure ChangeIntoName (var name : nameType; str : string);
    begin ... end ChangeIntoName;

    procedure ChangeIntoString (name : nameType; var str: string);
    begin ... end ChangeIntoString;
  end NameStorage;

  var data : array minAccountNo : maxAccountNo of
               record
                 name : nameType; balance : integer;
               end;
     availAccountNo : integer;

  procedure StoreName (acnt : integer; str : string);
  begin ChangeIntoName (data[acnt].name, str);
  end StoreName;

  procedure GetName (acnt : integer; var str: string);
  begin ChangeIntoString (data[acnt].name, str);
  end GetName;

  procedure GetNewAccount : integer;
  begin GetNewAccount := availAccountNo;
      inc (availAccountNo);
  end GetNewAccount;

  procedure AdjustBalance (acnt, amt : integer);
  begin inc (data[acnt].balance, amt);
  end AdjustBalance;

  procedure GetBalance (acnt : integer) : integer;
  begin GetBalance := data[acnt].balance;
  end GetBalance;
begin
  availAccountNo := minAccountNo;
end BookKeeper;
```

Figure 1-2. Outline of the BookKeeper module.

```
module RequestHandler;
  export ProcessRequest;
  import string, GetName, StoreName,
       GetNewAccount, AdjustBalance, GetBalance;

  type transType = (Deposit, Withdraw, Open, Print);

  module InputOutput;
    (* See Figure 1-1. *)
  end InputOutput;

  procedure PrintAccount;
  var name : string; acnt, amt : integer;
  begin
    ReadAccountNo (acnt); PrintAccountNo (acnt);
    GetName (acnt, name); PrintName (name);
    amt := GetBalance (acnt); PrintBlance (amt);
  end PrintAccount;

  procedure OpenAccount;
  var acnt : integer; name : string;
  begin
    acnt := GetNewAccount; ReadName (name);
    StoreName (acnt, name); PrintAccountNo (acnt);
  end OpenAccount;

  procedure ProcessRequest;
  var trans: transType; acnt, amt : integer;
  begin
    loop
      ReadTransType (trans);
      case trans of
        Deposit, Withdraw:
          begin ReadAccountNo (acnt); ReadAmount (amt);
              AdjustBalance (acnt, amt)
          end;
        Open: begin OpenAccount end;
        Print: begin PrintAccount end;
        otherwise: begin (* Print error messages *) end;
      end; (* case *)
    end; (* loop *)
  end ProcessRequest;
end RequestHandler;
```

Figure 1-3. Outline of the RequestHandler module.

```
(1) edit BookKeeper.AdjustBalance

(2) Modify the procedure to:

    procedure AdjustBalance (acnt, amt: integer);
    begin
      if (amt < 0) and (data[acnt].balance < -amt) then
         (* Print messages for overdraw *)
      elsif (amt > 0) and (maxInteger-amt < data[acnt].balance) then
         (* Print messages for exceeding the account limit *)
      else
         inc (data[acnt].balance, amt);
      end;
    end AdjustBalance;

(3) compile AdjustBalance

(4) update AdjustBalance
```

Figure 1-4. Modify AdjustBalance to check error conditions.

Let us assume that this banking system is currently running. Suppose that we want to modify the AdjustBalance procedure so that the system generates error messages when a balance becomes negative or exceeds the account's limit. Figure 1-4 shows how this modification can be carried out dynamically using our system.

Line (1) starts the editor for the AdjustBalance procedure. The editor gets a copy of the AdjustBalance procedure from the source program manager. The new version is shown in the Figure, where "maxInteger" is a built-in constant. After the AdjustBalance procedure has been modified, the new source code is saved for compilation by the source program manager.

The compile command at line (3) starts the recompilation of the AdjustBalance procedure with complete type checking. Type checking is carried out using the saved symbol tables from the previous version of the AdjustBalance procedure. The compiler generates the new object code and a new symbol table entry for the procedure.

The update command inserts the new object code into memory and modifies the current core image so that the new object code is executed by calls to the AdjustBalance procedure. Since the old object code might currently be executing, the old object code is only destroyed when it can no longer be referenced. That is, the current execution of the old object can continue even after the new object code is inserted into memory. When an update command is requested, our system checks whether the program corresponding to executable code after the update command is consistent. If not, the update command is not carried out.

1.4. Dissertation Plan

The organization of this dissertation is as follows: Chapter 2 reviews other systems that allow changes to a running program and explains how our system is different from them. Chapter 3 describes a base programming language and discusses assumptions and goals for our system. Furthermore, it describes the five components (command interpreter, source code manager, editor, compiler, and run-time support system) of the system. Chapter 4 presents and justifies dynamic modification features supported by our system. Chapter 5 develops a methodology that can be used to change a running program in incremental steps. Chapter 6 explains an implementation of our system and describes a run-time program representation that supports the dynamic modification of a program. Chapter 7 contains a summary and concludes the dissertation with further research problems.

CHAPTER 2

RELATED WORK

2.1. Introduction

There have been rather few published investigations on dynamic modification of programs [18,23,20]. We conjecture three probable reasons: first, there have been a limited number of applications for continuously running programs; second, there have been no machines or programming languages designed to support dynamic modification of programs; and third, people have limited their research to static modification, for example, using structured programming concepts to enhance ease of understanding, and therefore, ease of modification of programs.

This chapter presents a general review of systems that allow modifications to a running program. In the first two systems described below, changes to a running program are initiated as an external event; therefore, no prior consideration, during the development of a program, is required to allow dynamic changes to the program. However, in the third system, probable dynamic changes to a program have to be anticipated and built into the program since a running program can only be modified in a limited way.

2.2. A Data Base System

The problem of dynamically replacing a module in a data base system made up of many modules and processes, where each module manages its own data structures, has been studied by Fabry [Fab76]. His main concern was how to dynamically replace a module when the interface of the module was unchanged or merely augmented. He also considered how to detect and to update data structures whose type is different from that expected by the current code of the module.

In this system, changes to the data structures and code of a module are realized at different times. When the replacement of a module is requested, an indirect word containing the beginning address of the module's code segment is changed immediately to point to the new code segment. However, the module's data structures are not changed until the new code is executed.

To detect obsolete data or code, a current version number is stored with data structures and an expected version number is associated with code. When a module is executed, the current and expected version numbers are compared to see if the data or the code has become obsolete. The obsolete data is updated by conversion routines provided with the new version of the module. If the code has become obsolete (because new code and data have been installed while the two version numbers are compared), the new code, selected by the module's indirect word, is executed. These steps are repeated until the two version numbers match exactly.

He proposes that the system be built on top of an operating system that supports capability addressing described in [17]. The capability addressing provides process-independent interpretation of an address stored in an indirect word for a module so that all processes in the system can be affected by a single change. The disadvantage of his proposed implementation is that it requires the existence of a capability based operating system. Our system also allows several processes, but we assume that they share the same address space; therefore, it can be implemented with a simple indirect addressing scheme.

The major deficiency of his work is that the details are not examined. For example, it does not explain how a programmer can modify a running program.

2.3. An Experimental Operating System

An experimental operating system, called DAS (Dynamically Alterable System), that is composed of many modules and that allows dynamic replacement of one of the modules has been described in [23]. A module is implemented using code, object, and own segments, where each segment is addressed by a descriptor and descriptors are linked to represent the module. The code segment contains the code for procedures of the module and for operations on the user defined (abstract) data types within the module. The module's data is stored in two kinds of segments: the object segments contain data used by variables declared as user defined (abstract) data types and the own segment contains other local variables of the module. A procedure's local variables are located within a global stack segment.

As in Fabry's work, a module can be replaced only if its specification remains unchanged or is compatible with the old version. A module's code and own segments are replaced when a "replug" command is executed; but the replacement of an object segment is deferred until the old object segment is referenced. To detect an obsolete object segment, the object segment of a module is linked to the own (or code if no own) segment of the module and the linked own (or code) segment is compared with the current own (or code) segment of the module whenever the object segment is used. The dynamic replacement of a module is realized by creating a new code segment and data segments for the new version and by changing the segment descriptors of the module to point to the new segments.

DAS is the first implemented system that supports dynamic modification. The significance of DAS is that it has shown that it is feasible to build an operating system that can be modified dynamically. However, DAS has several limitations. First, only one module can be replaced at a time and the specification of the new version must be compatible to that of the old version. Furthermore, modules can not be added or deleted from DAS. These restrictions mean that DAS can be dynamically modified in a limited way. To support any changes to a running program, our system allows the addition, deletion, and replacement of several procedures and modules at the same time. Second, the arguments of user commands are segment descriptors. That is, the user commands are implementation-dependent. As we will see, our system is implementation-independent as far as the programmer is concerned. Finally, their work does not address the problem of how to compile a new module with type-checking or how to manage source code.

2.4. PROTEL

PROTEL (PRocedure Oriented Type Enforcing Language) is a high-level programming language designed at Bell-Northern Research for the development of software for large digital switching systems [20]. It is used for telephone switching systems that have to provide reliable uninterrupted service even when the systems are being reconfigured. To be able to dynamically modify programs written in PROTEL, it allows a new module to be added to a running program. Furthermore, it supports procedure variables so that a new module can be used from the current program by binding existing procedure variables to procedures of the new module. Such bindings occur within the "entry" procedure of the new module that is executed when the new module is added.

As an example of how a program can be modified, let us consider a file system that calls device-dependent open, close, get, and put procedures in an appropriate device-specific module. The file system maintains a table of device procedures that is indexed by device names. For example, the open file procedure might look as follows:

```
procedure openFile (fileName : string);
  ...
  (* The file is in the deviceName device *)
  deviceTable[deviceName].open (...);
  ...
```

When a new device module is added, it "binds" itself into the file system by calling a procedure in the file system with the device name and the open, close, get, and put procedures as arguments. The called procedure assigns the new device procedures to procedure variables in a corresponding device table entry.

In this system, a program can be dynamically modified only if the changes were anticipated when the program was initially developed. That is, arbitrary changes to a running program are not possible; for example, fixing a bug in the above open file procedure. However, PROTEL has still proven to be invaluable in developing and maintaining a family of telephone switching systems [32].

2.5. Summary

A survey of related work suggests that there have been limited attempts to design general-purpose dynamic modification systems that can support arbitrary changes to a running program. Our approach is similar to the first two systems described above in that changes to a running program are considered as external events to the running program. However, instead of designing a particular application system that can be dynamically modified, we provide an integrated programming system (that consists of a command interpreter, compiler, editor, source program manager, and run-time support system) so that all programs developed using our system can be modified dynamically. Furthermore, the following dynamic modification capabilities are supported by our system:

(1) Several procedures and modules can be modified simultaneously;

(2) Procedures and modules can be added or deleted;

(3) The specification of a procedure or module can be changed;

(4) Information stored in old data representations can be converted into new data representations immediately or later; and

(5) A programmer can specify when changes to a running program are to be carried out.

CHAPTER 3

THE SYSTEM

3.1. Introduction

The system supports the dynamic modification of a StarMod [8] program. We have chosen StarMod because it supports modular and concurrent programming. Our techniques should be applicable to other languages such as Modula [50], Mesa [38], or Ada [47].

The underlying reason for our assumptions and goals was that even if the current behavior of a running program is unacceptable, changes to the running program need not be made in "panic". That is, the changes to the running program should be carried out smoothly with as little interruption as possible. For example, if a program can be changed only when it is in a certain state, our system waits for that state instead of forcing the program to be in that state. Another underlying reason is that making changes to a running program is a continuing activity. Therefore, the system should support any and only changes that maintain the consistency of a program. The consistency is necessary for future changes. Finally, changes to a running program may cause unexpected events if the changes are carried out at an arbitrary moment. So the system should allow the programmer to specify when the program is to be changed.

In this chapter, we describe features of StarMod that are relevant to our work. Then, we discuss the assumptions and goals of our system. Finally, we provide a general overview of the five components of the system: a command interpreter, source code manager, editor, compiler, and run-time support system.

3.2. The Programming Language

A program is constructed by binding together one or more modules. Each module is composed of an export list, import list, constant definitions, type definitions, variable declarations, procedure definitions, and optional initialization code. A syntactic description of a module is as follows:

```
<module decl> ::= [ interface ] module <module id>;
                  [ <export list> ]
                  [ <import list> ]
                  { <const decls> | <type decls> | <var decls> |
                    <module decls> | <procedure and process decls> }
                  [ begin <stmt list> ]
                  end <module id>
```

A module is used to encapsulate the data structures, procedures, and processes necessary to implement an abstraction. Since modules can be separately compiled, there should be a predetermined "main" module in a program.

To prohibit simultaneous access to the data and procedures of a module, it can be declared as an interface module. The interface module is similar to Hoare's monitor [26]; that is, only one process can be executing within the interface module at a time.

The import list of a module specifies all identifiers of objects that are declared outside the module and that are to be visible within the module. The export list specifies all identifiers of objects that are declared within the module and that are to be visible outside the module. Exported types of a module are opaque; that is, their structural details are hidden. Therefore, variables of an exported type can only be manipulated by procedures of the module that defined the exported type. As we will see, this restriction is not required to support the dynamic modification of a module.

A procedure declaration consists of a procedure heading and body. The heading specifies the procedure identifier and the parameters for the procedure.

The body contains local declarations and statements. Modules can not be declared within a procedure, but they can be defined within other modules. The syntax of a procedure declaration is as follows:

```
<procedure> ::= procedure <proc id> ( [<params>] ) [: <type id>];
                  { <const decls> | <type decls> |
                    <var decls> | <procedure decls> }
                  begin [ <stmt list> ]
                  end <proc id>
```

Normal procedures do not return a result value and are activated by a procedure call. Function procedures return a result value and are called from expressions. There are two kinds of parameters, namely variable (i.e., reference) and value parameters. Variable parameters are indicated by the key word "var" in the formal parameter list. The variable parameters correspond to actual arguments; however, the change to the formal parameters need not be reflected to the actual arguments until completion of the procedure. That is, "var" parameters may not be implemented by passing an address. Value parameters act as initialized local variables.

The form of a process declaration is identical to that of a procedure with the key word "process" substituted for "procedure". Process declarations are allowed in any modules that are not nested in an interface module. A process is activated by a process call statement that is syntactically identical to the procedure call statement. However, the execution of the calling program continues in parallel with the new process. Process call statements are restricted to the bodies of modules; that is, they can neither occur within procedures nor processes.

Other language constructs, such as built-in data types and statements, are similar to those of Modula [50] and will not be detailed. The complete definition of the language can be found in [6].

3.3. Assumptions and Goals

We assume that a basic dynamic modification unit is a procedure or module. A procedure is chosen because it is a basic unit for code abstraction. Furthermore, analysis of Pascal [9] and SAL [44] programs shows that most procedures are rather small; for example, 70 percent of the Pascal procedures contained less than or equal to 16 statements and the average SAL procedure had 18.2 statements. Therefore, the replacement of a procedure to change a few statements within it seems reasonable. A module is chosen because it is a basic unit for data abstraction. Also, procedures and modules are the usual units of separate compilation. We note that our approach could be extended to allow modification to an arbitrary statement of a program by using threaded code [30].

Even when a procedure that has called another procedure is replaced, our approach does not permit a return address stored in an activation record to be modified. This restriction is necessary because of the difficulties involved in specifying how the call statements of the old version match those of the new version. Therefore, when a procedure is replaced, the old code is not destroyed so that processes executing the old version can *continue* their execution.

Because of the above approach, activations of both the old and new versions of a procedure may coexist. However, the coexistence of activations of both versions of some procedures might lead a program into an inconsistent state. So we allow the programmer to specify procedures that should be replaced only when they are not executing. For such procedures, activations of either the old or new, but not both, versions can exist at any one time. In either case, the new version will be used by the calls that are executed after the replacement.

Since the programmer can not know exactly which part of a program will be executing when the program is changed, changes to a running program can cause the program to behave unexpectedly. For example, a process may invoke a new procedure, instead of an old procedure, with a wrong number of arguments. Therefore, our system allows the programmer to define when a program can be modified; that is, to specify procedures that should not be executed during the modification.

Four goals were maintained in designing and implementing the dynamic modification system.

(1) Only syntactically and semantically correct modifications are allowed to ensure that the source code really represents the current executable code in memory.

(2) No prior consideration is required for a program to be dynamically modifiable. This goal allows any expected or unexpected changes to a program to be carried out dynamically.

(3) Each modification objective should be realizable with a minimal change to a running program to make our system easier and more practical to use. For example, replacing several modules to modify a few procedures is not acceptable.

(4) Overhead to a running program should be small, especially when dynamic modification is not requested, since dynamic modification to a program is a rare event. Furthermore, deleted code and data space should be recovered.

3.4. System Overview

The DYnamic MOdification System (DYMOS) consists of a command interpreter, source code manager, editor, compiler, and run-time support system. Figure 3-1 shows the overall structure of the system.

3.4.1. The Command Interpreter

The command interpreter receives a request from the programmer and then starts an appropriate component of the system (see Appendix A for the syntactic description of user commands). There are three main commands: **edit** to modify and create a procedure or module, **compile** to generate new object code, and **update** to insert new object code into memory. The usual dynamic modification session consists of a sequence of edit and compile commands followed by one update

```
                                         +-------------------+
                                    |==>|Source Code Manager|
                                    |    +-------------------+
P                                   |
R                                   |      +-------+
O                                   |===>|Editor|
G       +------------+              |      +-------+
R  =>   |   Command   |==>|
A       |Interpreter|                 |      +---------+
M       +------------+              |===>|Compiler|
M                                   |      +---------+
E                                   |
R                                   |     +-----------------------+
                                    |==>|Run-Time Support System|
                                          +-----------------------+
```

Figure 3-1. The overall structure of the system.

command.

After the modification and compilation of procedures and modules, the new code and data can be inserted into the currently running program by an update command. The update command has the following format:

update <arg list1> [**delete** <arg list2>] [**when** <arg list3> **idle**]

<arg list> has the following format:

<arg list> ::= <arg> { , <arg> }

where <arg> is a procedure or module name. If the procedure or module name is not unique in a program, the name can be qualified by its enclosing procedure or module name with a '.' separating any two names until it becomes unique. When an update command is requested, the system waits until the procedures specified in <arg list3> are not executed. Then, the system replaces the old versions of the procedures and modules in <arg list1> and makes the object code of the procedures and modules in <arg list2> unavailable for execution. The replacement and deletion are carried out as an indivisible operation. Furthermore, the when-condition is maintained while the procedures and modules are replaced and deleted. As an example, suppose we have module M and procedures P, Q, and R. Procedures P and Q and module M can be replaced and procedure R can be deleted when procedures P, R, and M.T are not executing by the update command

update P, Q, M **delete** R **when** P, R, M.T **idle**

M.T is procedure T of module M.

The procedures and modules specified in <arg list1> are replaced as an indivisible operation; that is, processes can not use any of the procedures and modules in <arg list1> while they are all replaced. This restriction is necessary because procedures and modules are updated together to ensure that their old versions can not be used after the replacement has begun. In the previous example, suppose the new version of procedure P calls procedure Q. Since procedures P and Q are replaced indivisibly, the new version of procedure P can not call the old version of Q.

If the update command contains the optional "**delete** <arg list2>", the procedures and modules in <arg list2> can no longer be used. The code and data space used by the procedures and modules are reclaimed immediately so that they can no longer be referenced. Although the procedures and modules are deleted as an indivisible operation, the correctness of the deletion should not depend on the indivisibility. That is, the programmer needs to ensure that procedures and modules are no longer needed before deleting them. For example, procedure R is specified in the when-part to ensure that it is not used when it is deleted.

The optional when-part can be specified to ensure that the replacement and deletion take place when the program is in an expected state; that is, when the procedures specified in <arg list3> are not executed. Such procedures are not available for execution until the replacement and deletion are completed. However, the procedures in the when-part can be called as long as any of them are being executed since those procedures may call other procedures in the when-part to finish their execution. Furthermore, the procedures specified in the when-part should not be unnecessarily blocked while waiting for the when-condition to be satisfied. We note that the procedures in <arg list3> do not have to be included in <arg list1> or <arg list2>.

It is possible that some when-condition's can be satisfied sooner if procedures in the when-part can not be called while the system is waiting. In the worst case, a when-condition might be satisfied only if procedures in the when-part are no longer called while the system is waiting. However, we believe that changes to a running program should be carried out with as little interruption as possible to the running program. In particular, a running program should not be stopped for modification (however, this feature could easily be added to our system). To prevent the system from waiting indefinitely, the system is timed out if the when-condition is not satisfied within some fixed time limit.

Since our goal is to let processes executing the old version to continue their execution, the old object code of replaced procedures is not removed until it can no longer be referenced. We note that if a procedure is specified in both <arg list1> and <arg list3>, activations of both the old and new versions of the procedure can never coexist. We could also let processes continue their execution of deleted procedures and modules as is done for the replaced procedures; however, we could not find any useful examples that required the delayed recovery of the code and data space for deleted procedures and modules.

Definition: We say that a program is *consistent* if it is syntactically and semantically (that is, compile-time) correct.

For example, if some portions of a program reference an array variable as record, the program is inconsistent. To maintain the consistency of a program, the command interpreter checks whether the program corresponding to executable code after the current update command is consistent. If the program might become inconsistent, the update command is not carried out. For example, suppose the new version of a procedure contains an extra parameter. If the users of the procedure are not modified to provide an extra argument, the program becomes inconsistent after the procedure is updated. Therefore, the update command is not carried out.

3.4.2. The Source Code Manager

The source code manager retrieves and stores the source code of a procedure or module on a request from the command interpreter. A source program is stored as a tree that preserves the hierarchical structure of the program. Each node of a source code tree represents a module or a procedure. The source code tree is created when the program is compiled for the first time and is modified whenever the new version of a procedure or module is created or updated.

Each node of the source code tree can contain both the current and the provisional versions of source code. The current version always corresponds to the executable code in memory. The provisional version of a procedure or module is created whenever the procedure or module is modified or created using the editor; the provisional version becomes the current version when the object code of the provisional version is updated. When the source code of a procedure or module is requested from the command interpreter (for the editor or compiler), the nested procedures and modules of the requested procedure or module are also returned. Whenever the command interpreter requests the source code of a procedure or module (for the editor or compiler), the source code manager returns the provisional version of the procedure or module if it exists; otherwise, the current version is returned. The nested procedures and modules of the requested procedure or module are also returned.

3.4.3. The Editor

The editor is an ordinary text editor; for example, "ed" or "vi" on Unix. It is used to modify or to create a procedure or module. To modify a procedure or module, the programmer starts an editing session by the command

edit <arg> [**from (current | provisional)**]

<arg> is the name of the procedure or module to be modified. As before, qualification can be used to resolve ambiguity. If the argument names an existing procedure or module, the editor can be started with either the current or the provisional version of the source code of the argument. If the optional "**from current**" is specified, the editor is started with the current version. If the optional "**from provisional**" is specified, the editor is started with the provisional version. If the optional from-part is not specified, the editor is started with the provisional version unless it does not exist, in which case the editor is started with the current version. If the argument does not name an existing procedure or module, a new procedure or module is to be created. Here the editor is started with an empty source code.

When the editing session is completed, the new version is stored as a provisional version in the source code tree. Since only one provisional version can be stored in a node, the new provisional version replaces the old one if necessary. We note that the provisional version can be incomplete or incorrect.

3.4.4. The Compiler

The compiler accepts the source code of a procedure or module and generates object code and symbol table information. The compiler allows separate compilation of modules and recompilation of a procedure or a module within a separately

compiled module. To support type checking, the compiler reads in the necessary symbolic information from the symbol table data base before compilation and writes out the new or changed symbolic information after compilation. The command

compile [<arg1>] [(**before** | **after**) <arg2>]

compiles the new version of a procedure or module <arg1>. If <arg1> is omitted, the procedure or module last edited is compiled. If <arg1> has been previously compiled, the new version is compiled using the saved compile-time environment of the old version.

If <arg1> is a new procedure or module, its environment must be specified by <arg2>; that is, <arg1> is compiled in the environment of <arg2>. The "**before** <arg2>" (or "**after** <arg2>") specifies that the source code of the new procedure or module be stored in the source code tree as if it had appeared before <arg2> (or after <arg2>). Both "**before**" and "**after**" are necessary to be able to place <arg1> at the beginning and at the end, respectively, within the construct that is to contain <arg1>. For example, a new procedure P of module M can be compiled as if it appeared after procedure Q of module M by the command

compile P **after** M.Q

The source code of procedure P is stored after that of procedure Q within module M.

When a module is recompiled, the default option is to assign variables and procedures of the new version the same addresses as those of the old version whenever possible. Here procedures and modules that only use unchanged variables and procedures of the module are not affected by the recompilation. Alternatively, the variables and procedures of the new version can be assigned addresses as if the module were being compiled for the first time. Here all procedures and modules that use the recompiled module should be (modified and) recompiled to maintain the consistency of the program. However, when a procedure is recompiled, the same address is always assigned to the procedure, but the addresses of its local variables are assigned independent of the previous assignments since they can not be shared.

Sometimes the definitions of procedures and modules need to be deleted from the symbol table data base so that other procedures and modules can be recompiled without them. A special command to modify the symbol table data base is provided. The command

delete <arg list>

deletes the definitions of the procedures and modules specified in <arg list> from the symbol table data base.

3.4.5. The Run-Time Support System

The run-time support system consists of the dynamic modification and the garbage collection processes. When an update command is issued, the command interpreter sends a request to the dynamic modification process that loads the new object code into the available memory space. The dynamic modification process detects when the procedures specified in the when-part of an update command are not executed. It then modifies the current core image to incorporate the new object code indivisibly (e.g., changing an indirect address word to point to the new object code) with the when-condition of the update command sustained. Instead of waiting indefinitely for the when-condition, the dynamic modification process aborts and returns a failure message to the command interpreter if the when-condition is not met within a programmer specified fixed time limit. Such time limit can be specified within the when-part of an update command as follows:

when <arg list> **idle** [**within** <limit>]

<limit> is a real number and represents seconds. For example, after the command

update P **when** P, Q **idle within** 10.0

is issued, if procedures P and Q do not become idle within 10 seconds, the command interpreter generates a timeout error message to the programmer. The programmer then may issue a new update command with a different time limit or when-part.

When modification is completed, the dynamic modification process sends a message specifying which parts of the memory can be reclaimed to the garbage collection process. The garbage collection process reclaims the parts of the memory when they can not be referenced by user processes any more through updating the available memory data base.

The dynamic modification and garbage collection processes are executed in parallel with other currently active processes.

3.5. Summary

The system supports the dynamic modification of programs written in a single language, in our case StarMod. The system provides an integrated environment that supports the editing, source code maintenance, compilation, and dynamic modification of a StarMod program. To provide a unified view of the system and the supported programming language, procedures and modules are identified by their names within the system.

Although the source code tree and the symbol table data base are described as separate entities, they are all part of a central data base, called the program structure tree (to be described in Section 6.2.1). The program structure tree contains, for each procedure and module, the current and provisional source code, object code, symbol table, and export and import lists. The command interpreter use it to check the validity of arguments, to ensure the consistency of a program after an update command, and to supply source code to the editor and compiler. The compiler use it to rebuild a compile-time environment for type checking.

CHAPTER 4

DYNAMIC MODIFICATION CAPABILITIES

4.1. Introduction

When programs are modified for various reasons, such as to fix bugs, to add and to delete features, to make improvements, and to adjust to new environments, the changes are ultimately carried out on running programs. To be able to use our system for all these purposes, the system is designed to support any changes to a running program as long as the program remains consistent.

This chapter describes the dynamic modification features supported by our system and explains why they are necessary. Sections 4.1 and 4.2 explain the dynamic modification features for procedures and modules, respectively. Section 4.3 discusses how to convert information stored in an old data representation into a new data representation when a module is replaced.

4.2. Modifying Procedures

This section describes how procedures can be replaced, added, and deleted from a running program. Since part of a program may become inconsistent when procedures are replaced or deleted, we explain how to specify that the inconsistent portion is not to be executed until the inconsistency has been removed. If the parameters of a procedure are redefined, the callers of the old version become inconsistent. However, it may not be possible to modify all the callers of the old version to satisfy the new parameters. We describe how the new version with changed parameters can replace the old version without affecting its callers. As we said in Chapter 3, our approach is that processes executing the old version continue their execution when a procedure is replaced. Since the new version of a procedure is used by calls that are executed after the modification, replacing a procedure that continuously loops can not be handled without additional mechanisms. We discuss how to transfer execution from the old version to the new version.

4.2.1. Replacing Procedures

A single procedure can be modified, recompiled, and updated as shown in Section 1.2. If a procedure contains nested procedures, the nested procedures are automatically included in each step of the dynamic modification. For example, suppose we want to modify procedure P, where procedure P contains procedure Q. Both procedures, P and Q, can be modified by editing, recompiling, and updating procedure P. Procedure Q is automatically included in all the steps since it is nested within procedure P.

There are cases where several, not necessarily nested, procedures need to be updated at the same time. For example, if the new version of a procedure must call the new version of another procedure, the two procedures need to be replaced together. When several procedures are modified, they should be compiled in the correct order so that later compilations use the new definitions. Furthermore, when they are updated, proper conditions should be specified to prevent processes from executing the old version of an updated procedure. As we explained in Section 3.4.1, if several procedures are updated together, they are replaced indivisibly; that is, processes can not call them until all of them are replaced.

As an example of how to replace several procedures, suppose we want to change the GetBalance procedure in Figure 1-2 to return both the name and the balance of a given account. Figure 4-1 shows how this modification can be carried out in our system. Since the PrintAccount procedure calls the GetBalance

```
(1) edit GetBalance

(2) Modify the procedure to:

   procedure GetBalance (acnt: integer; var name: string; var amt: integer);
   begin
     GetName (acnt, name); amt := data[acnt].balance;
   end GetBalance;

(3) compile

(4) edit PrintAccount

(5) Modify the procedure to:

   procedure PrintAccount;
   var name: string; amt: integer;
   begin
      ReadAccountNo (acnt); PrintAccountNo (acnt);
      GetBalance (acnt, name, amt);
      PrintName (name); PrintBalance (amt);
   end PrintAccount;

(6) compile

(7) update GetBalance, PrintAccount when PrintAccount idle
```

Figure 4-1. Modify GetBalance and PrintAccount simultaneously.

procedure, the PrintAccount procedure is also modified to use the new GetBalance procedure and then recompiled after the GetBalance procedure has been recompiled. If either the GetBalance or the PrintAccount procedure is updated first, the program becomes inconsistent. For example, if the PrintAccount procedure is replaced first, then the new version may call the old version of the GetBalance procedure. Conversely, if the GetBalance procedure is replaced first, then the old version of the PrintAccount procedure may call the new version of the GetBalance procedure. Since such calls should be prevented, the two procedures have to be updated together. Furthermore, if the old version of the PrintAccount procedure is being executed when they are replaced, the old version may, as before, call the new version of the GetBalance procedure. Therefore, we need to tell the system to replace two procedures when the old version of the PrintAccount procedure is not executed. That is, the PrintAccount procedure should be specified in the when-part of the update command in line (7). That update command says that the two procedures must be replaced while the PrintAccount procedure is not executed.

4.2.2. Adding Procedures

Procedures can be added to the current program after they are created with the editor and then compiled. To create a new procedure, say P, the programmer issues the command

edit P

then enters the source code of procedure P using the editor. We note that the identifier P should not match any existing procedure or module name. If necessary, P can be qualified by the name of a procedure or module that is to contain P. To compile the new procedure P, its environment is specified in the compile command as before (or after) an existing procedure or module, say A, as follows:

compile P **before** A

The source code of procedure P will be stored before the source code of A in the source code tree. The name of a new procedure should not be already visible within the scope that is to contain the new procedure to guarantee that the program is consistent after the new procedure is added.

As an example of how to add procedures, let us add a new procedure ProcessTrans that adjusts the balance of an account by a given amount for a Deposit or WithDraw request before the ProcessRequest procedure in Figure 1-3. Figure 4-2 shows how this addition can be carried out in our system. Note that the compile command at line (3) contains the "**before** ProcessRequest" part to provide

```
(1) edit ProcessTrans

(2) Create the procedure:

   procedure ProcessTrans (trans: transType; acnt, amt : integer);
   begin
      if amt < 0 then
         (* Print error messages *)
      elsif trans = Deposit then
         AdjustBalance (acnt, amt);
      else
         AdjustBalance (acnt, -amt);
      end;
   end ProcessTrans;

(3) compile ProcessTrans before ProcessRequest

(4) update ProcessTrans
```

Figure 4-2. Add ProcessTrans before ProcessRequest.

an environment necessary for the compilation and to specify that the ProcessTrans procedure is to be placed before the ProcessRequest procedure in the source code tree. The update command does not contain a when-part since processes can not yet execute the new procedure.

4.2.3. Deleting Procedures

Procedures can also be deleted from the current program. The deletion of procedures from a running program is usually carried out in three steps. First, their definitions are deleted from the symbol table data base to make them unavailable to subsequent compilations as follows:

delete <arg list>

<arg list> is the list of procedures to be deleted. This step is not always required but is necessary if other objects with the same names are going to replace the deleted procedures. For example, suppose module M contains procedures P and Q.

Furthermore, procedure P also contains procedure Q; that is, there are two procedures with the name, Q. Suppose other procedures nested within procedure P need to use procedure Q within module M instead of procedure Q nested within procedure P. Here the definition of procedure Q nested within procedure P has to be deleted before other nested procedures are recompiled.

Next, other procedures are modified and compiled so that they no longer use the deleted procedures. Finally, the procedures are deleted and replaced in the running program by the command

update <arg list1> **delete** <arg list2> **when** <arg list3> **idle**

<arg list1> contains the procedures that are modified because of the deleted procedures in <arg list2>. <arg list3> specifies when <arg list1> and <arg list2> should be replaced and deleted; for example, when the procedures in <arg list1> and <arg list2> are not in use.

It would be desirable if procedures could always be replaced and deleted in two separate commands. That is, it would be simpler to modify a program if the procedures in <arg list1> are first replaced and then the procedures in <arg list2> are deleted, but they never need to be replaced and deleted at the same time. However, the deletion of procedures can not always be delayed until all procedures that use the deleted procedures are replaced. Therefore, the update command can not be replaced by two different commands: one for deletion and another for replacement and addition.

As an example of when procedures have to be replaced and deleted together, suppose we have three procedures P, Q, and R, where procedure Q calls procedure R and procedure P is the only procedure that calls procedure Q. We assume that we do not know which procedure is currently being executed. Suppose we delete procedure Q and modify procedures P and R, where procedure R is modified to have different parameters and procedure P is modified to call procedure R. After the recompilation of procedures P and R, procedures P, Q and R have to be replaced and deleted together as follows:

update P, R **delete** Q **when** P **idle**

to maintain the consistency of the program. The reasons why they have to be replaced and deleted together are as follows:

(1) Since the old version of Q can not call the new version of R, the deletion of Q should be done before or together with the replacement of R. Furthermore, there should be no outstanding activations of Q when P is replaced.

(2) Since the old version of P calls Q, Q can not be deleted until P has been replaced. Furthermore, there should be no outstanding activations of the old version of P when Q is deleted.

(3) Since the new version of P calls the new version of R, the replacement of P should not be done until R has been replaced.

Therefore, procedures P and R should be replaced and procedure Q should be deleted at the same time when procedures P and Q are not in use. Since procedure P is the only procedure that can call procedure Q, procedure Q is not specified in the when-part.

The garbage collection process can immediately reclaim the space used by the deleted procedure Q and the old version of procedure P since procedure Q and the old version of procedure P can no longer be used. The space used by the old version of procedure R will be reclaimed when processes can no longer use the old version of procedure R.

We note that any procedures can be deleted from a running program if other procedures that call the deleted procedures can be modified so that they do not call the deleted procedures.

4.2.4. Redefining Parameters

Sometimes it is necessary to modify more than just a procedure body. For example, we may include an extra parameter to increase the functionality of a procedure as was done in Figure 4-1. When a procedure's parameters are redefined, the consistency of a program can be maintained as follows: All procedures that call the procedure are modified to include new arguments and recompiled. They are then updated with the procedure at the same time as discussed in the section on "Replacing Procedures". Here, they should be specified in the when-part to prevent their old versions from calling the new version of the procedure with incorrect arguments.

Instead of modifying other procedures to supply correct arguments, the new version can provide a *parameter convert* routine (called convert routine for short) that dynamically transforms the actual arguments of the old version into those required by the new version. The convert routine is executed only for calls to the old version and its presence is transparent to users of the procedure. The syntactic description of a new procedure with the convert routine is as follows:

```
<new proc decl> ::= procedure <proc id> [ (<param>) ] [: <type id>]
                      [ convert
                          [ <var decls> ]
                          [ before <stmt list> ]
                          [ after <stmt list> ]
                        end; ]
                      [ <declarations> ]
                      begin
                          [ <stmt list> ]
                      end <proc id>;

  <old param ref> ::= <old param id> |
                      <proc id> . <old param id>

<old return value id> ::= <proc id>
```

Within the convert routine, the parameters of both the old and the new versions can be referenced. The "var" parameters of both versions of the procedure are treated as pointer variables and the built-in function "addr" returns the address of an argument. References to an old parameter can be qualified by the procedure name to resolve ambiguity.

After the new version with the convert routine is updated, when a call to the old version is executed, the before-part of the convert routine is executed to generate and to initialize the arguments to the new version. For example, if a new parameter has no corresponding old parameter, it can be assigned a default value in the before-part. Since "var" parameters are treated as pointer variables, the "var" parameters of the new version should be assigned addresses of variables that are to be used as actual arguments to the new version. For example, if a "var" parameter of the old version is assigned to a "var" parameter of the new version, an actual argument to the new version is the same as that of the old version. After the execution of the new version, the after-part is executed to assign values to the actual arguments of the original call. If the old version is a function procedure, a return value to the old version can be assigned within the after-part of the convert routine by assigning a value to the procedure name.

To illustrate how to use a parameter convert routine, let us modify the GetBalance procedure as in Figure 4-1 but not the PrintAccount procedure. Since the parameters of the old and new versions of the GetBalance procedure do not match, the new version contains the convert routine that maps the parameters of the old version into the parameters of the new version (see Figure 4-3). Since the old version does not have parameters corresponding to the parameters name and amt, the convert routine declares the local variables, t_name and t_amt, that are to be used as the actual arguments for name and amt. The parameter acnt of the new version is initialized by that of the old version in the before-part and the return value to the old version is assigned in the after-part. Because of the parameter

```
(1) edit GetBalance

(2) Modify the procedure to:

   procedure GetBalance (acnt: integer; var name: string; var amt: integer);
   convert
      var t_name : string; t_amt : integer;
      before
        name := addr (t_name); amt := addr (t_amt);
        acnt := GetBalance.acnt;
      after
        GetBalance := t_amt;
   end;
   begin
       GetName (acnt, name);
       amt := data[acnt].balance;
   end GetBalance;

(3) compile

(4) update GetBalance
```

Figure 4-3. Modify GetBalance without modifying PrintAccount.

convert routine, the PrintAccount procedure needs not be modified to call the new version of the GetBalance procedure (as was done in Figure 4-1).

Since only the new version of a procedure is available for compilation, procedures should be modified to call the new version before they are recompiled. The code space used for the parameter convert routine and the old version is reclaimed when all references to the old version have been modified to call the new version.

We note that having a parameter convert routine within the new version does not increase the procedure replacement capability. The same effect can be achieved by making the new version a new procedure. The old version is then modified to call the new procedure with proper arguments. However, the use of a parameter convert routine does not increase the complexity of procedure call relations. Furthermore, since the definition of the old version is not available, it forces the programmer to modify procedures that called the old version to call the new version whenever such procedures are recompiled.

4.2.5. Non-delayed Replacement

The new version of a procedure is used by calls that are executed after the modification. But sometimes that approach is not feasible. Suppose we have replaced a procedure that continuously loops. If the procedure is never called again, the new version will never be used unless execution can transfer directly from the old version to the new version. Here the procedure should not be specified in the when-part of an update command.

A *labeled* statement and a *before-label* convert routine within the new version provide the appropriate semantics. The label statement defines the statement in the old version from which control can transfer to the new version. The before-label convert routine initializes the local variables of the new version after the transfer. To include these features, the "convert" section is augmented as follows:

```
<before-label> ::= convert
                     [ <var decls> ]
                     { before <label> <stmt list> }
                   end;

<labeled stmt> ::= <label> <stmt>
       <label> ::= '<<' char string '>>'

 <old local ref> ::= <old local id> |
                     <proc id> . <old local id>
```

<label> is a unique character string that identifies a statement within the old version. The scope of the convert routine includes the local variables of both the old and new versions of the procedure. As before, references to old variables can be qualified to resolve ambiguity. However, unlike the previous convert routine, the "var" parameters are treated as ordinary variables. The local variables of two

versions need not be the same; however, the parameters of both versions should be the same.

After the new version has been updated, when execution reaches any statement that is used as a label in the new version, execution continues from the labeled statement in the new version. However, before execution resumes at the labeled statement, the old activation record is converted to the format required by the new version. Then, the matching before-label statement is executed to initialize the local variables of the new version. Note that since the before-label routine can reference local variables of the old version, the old activation record should not be destroyed until the initialization is completed.

```
procedure ProcessRequest;
var trans : transType; acnt, amt : integer;
convert
   before <<case trans>>
      (* case statement in the previous version *)
      trans := ProcessRequest.trans;
      (* acnt and amt need not be restored *)
end;
begin
   loop
      ReadTransType (trans);
    <<case trans>>
      case trans of
         Deposit, Withdraw:
            begin ReadAccount (acnt); ReadAmount (amt);
                ProcessTrans (trans, acnt, amt);
            end;
         .
         (* This part is not changed. *)
         .
      end; (* case *)
   end; (* loop *)
end ProcessRequest;
```

Figure 4-4. Modify ProcessRequest to use new input formats.

As an example of a non-delayed replacement, let us modify the ProcessRequest procedure in Figure 1-3 to take a positive amount for the Deposit and Withdraw requests and to call the ProcessTrans procedure added in Figure 4-2. Figure 4-4 shows the new version of the ProcessRequest procedure. After the new version has been updated, when execution reaches the case statement of the old version, the "**before** <<**case** trans>>" statement is executed to initialize the variable trans of the new version to the value returned from the ReadTransType procedure within the old version. Then, the case statement of the new version is executed. Note that if "<<ReadTransType>>" is used as a label instead of "<<**case** trans>>", the convert routine is not needed since a value of the variable trans of the old version needs not be remembered.

Since active processes can be thought of as procedures that are continuously executed, they can also be replaced with this method. The before-label code can be garbage collected when processes can no longer use the old version of a procedure.

4.3. Modifying Modules

This section describes how modules can be added, deleted, and replaced in a running program. Since the running program should always be consistent, modules can be added only if they do not cause inconsistency. We note that this restriction does not limit the modules that can be added because the names of the exported objects of new modules can be chosen to avoid the inconsistency. However, when modules are deleted, other procedures and modules need to be modified and updated to ensure that the resulting program is consistent.

The replacement of a module is called either *transparent* or *visible* depending on whether other parts of a program become inconsistent when the new version is compiled. That is, if the new version can replace the old version without additional changes to other parts of the program, the replacement is transparent; otherwise, the replacement is visible. To reduce the ripple effect from recompilation of modules, exported variables and procedures that are not changed are assigned to the same addresses. If the replacement of a module is visible, additional modules and procedures need to be modified and updated to maintain the consistency of a program.

In our system, if the program becomes inconsistent, the inconsistency can be removed with the minimum recompilation of the affected procedures and modules. For example, if procedures, but not the data representation, of a module become inconsistent, the inconsistency is resolved by modifying and recompiling those procedures only; that is, the whole module need not be recompiled. We define the procedures and modules that can become inconsistent when a module is recompiled and explain how the inconsistency can be resolved.

4.3.1. Adding Modules

Modules can be added to the current program after they are created with the editor and then compiled. As before, when the compilation of a new module is requested, its environment needs to be specified in the compile command. To guarantee that the program will be consistent after the new module is added, the compiler checks that the declaration of the new module is valid in the scope. That is, the names of the module and its exported objects are not already visible in the scope that is to contain the module. This restriction implies that if the new module is to export an object whose name is already visible within the enclosing scope, the object with the same name should be made invisible prior to the addition of the new module.

When a new module is updated, its initialization statements are executed. The initialization statements may reference imported variables. If the values of such imported variables should not be changed while the initialization statements are executed, procedures that can assign values to them should not be executed when the module is added. That is, such procedures need to be specified in the when-part of an update command for the module.

As an example of how to add modules, suppose we want to add module N after module M in Figure 4-5 (a). Let us assume that two procedures, P and Q, are the only procedures that assign values to variables x and y. After module N has been created as in Figure 4-5 (b), it can be compiled as if it appears after module M of module MN by the command

compile N **after** MN.M

If the values of variables x and y should not be changed while module M is initialized, the module can be added by the command

(a) Outline of the existing module MN.

```
module MN;
  export x,y,...;
  ...
  var x,y ... ;
  procedure P;  (* use x *)  end P;
  procedure Q;  (* use y *)  end Q;
  module M;
    ...
  end M;
end MN;
```

(b) Outline of the new module N.

```
module N;
  import x,y;
  ...
begin
  (* use x,y *)
end N;
```

Figure 4-5. Outline of the modules MN and N.

update N **when** MN.P, MN.Q **idle**

Since procedures P and Q are the only procedures that assign values to variables x and y, the values of variables x and y will not be changed while the module is initialized.

4.3.2. Deleting Modules

Modules can also be deleted from a running program. As with procedures, modules are deleted from the current program in three steps. For example, suppose we want to delete the module M in Figure 4-6. Let us assume that the exported type t of module M is only used within module N and procedure R. The definition of module M is first deleted from the symbol table data base. Then, module N and procedure R are modified and recompiled without the definition of module M. Finally, module N and procedure R are replaced and module M is deleted at the same time as follows:

update N, R **delete** M **when** M.P, Q, R **idle**

Procedure P of module M is specified in the when-part to ensure that module M is

```
module M;
  export t, P;
  (* t is a type and P is a procedure *)
  ...
end M;
module N;
  export Q;
  (* Q is a procedure *)
  import t, P;
  ...
end N;
procedure R;  (* call P *)  end R;
```

Figure 4-6. Outline of module M and its users.

not used when it is deleted. Procedures Q and R are also specified to ensure that module N and procedure R are not used when they are replaced.

4.3.3. Replacing Modules

In our system, if a programmer wants to change anything other than procedures, modules have to be replaced because procedures and modules are the basic units of modification. Instead of replacing both the object code and the data of a module whenever the module is updated, we allow the module to be updated to replace:

(1) the symbol table definitions;

(2) the symbol table definitions and the object code; or

(3) the symbol table definitions, the object code, and the data;

Other combinations are not meaningful. For example, the module can not be updated to replace only the symbol table definitions and the data since if the data representation of the module is changed, its code also has to be changed to use the new data representation.

To generate only the changed components of a module, the compile command for the module is extended as follows:

compile <module id> [**for (definition | code)**]

If the "**for definition**" option is specified, only the new symbol table definitions are generated. Here the object code and the data representation of the new version (if generated) should be the same as those of the old version. If the "**for code**" option is specified, the symbol table definitions and the new object code are generated. Again, the data representation of the new version should remain unchanged from that of the old version. Otherwise, the new symbol table definitions, the new object code, and the new data for the module are generated.

We note that this process of replacing only the changed components can be simplified; that is, the compiler can generate all three components and then decide which components are changed from the previous version.

Whenever a module is recompiled, the module must be consistent regardless of which option is used. We could let only the declaration part of a module be processed when the module is recompiled with the "**for definition**" option. Then, the inconsistent procedures of the module can be modified and recompiled separately. This approach was not chosen because our language does not explicitly distinguish between the declaration part (that is, everything except procedure bodies and initialization statements) and the code part of a module.

4.3.3.1. Transparent Modifications to a Module

Some transparent modifications to a module require changes to the definitions stored in the symbol table data base for the module but not the object code or the data representation. Although such modifications do not affect the currently running program, they may be necessary for future changes. These modifications consist of one or more of the following cases:

(1) An identifier is added to the export list; the added identifier should be visible within the module and should not cause a naming conflict within the enclosing scope.

(2) An identifier is deleted from the import list; the deleted identifier should not be used within the enclosing scope.

(3) An identifier is added to the import list; the added identifier should be visible in the enclosing scope and should not cause a naming conflict within the module.

(4) An identifier is deleted from the import list; the deleted identifier should not be used within the module.

(5) A constant or type definition that is neither used nor exported from the module is changed.

The new symbol table definitions for the module are generated by compiling the new version with the "**for definition**" option.

There are transparent modifications to a module that require changes to the symbol table definitions and the object code, but not the data representation. For example, if a regular module is changed to an interface module (or vice versa), the change is transparent to its users and the data representation need not be modified. If the on-line banking system described in Section 1.2 is to be expanded to handle simultaneous requests, the BookKeeper module needs to be changed to an interface module. Otherwise, two Withdraw or Deposit requests to the same account can be handled in an interleaved fashion with the incorrect resulting effect. Another example is that if the new version changes the value of a constant or the structure of a type that is used only within the procedure bodies of a module, the data representation is not affected by the change. As an optimization, procedures whose object code is unchanged are not replaced when the module is updated.

The procedures to be specified in the when-part of an update command depend on the kind of changes made to the module. For example, if a regular module is converted to an interface module, there should be no processes already executing within the interface module when the conversion is completed. That is, the module should not be executed when it is converted. This restriction can be enforced by specifying all the exported procedures in the when-part.

Finally, the other transparent modifications to a module require changes to the internal data representation, and therefore, the symbol table definitions and the object code. For example, suppose a module implements a stack and the operations, push and pop. We assume that the stack is represented by an array. If the module is changed to use a linked list to represent the stack, the procedure bodies should be modified accordingly; but the users of the stack need not be modified or recompiled since the type of the operations are not changed. We note that modifications to the internal data representation can be transparent because the exported variables and procedures are assigned the same addresses.

If a module's data representation is changed, the module (that is, its exported procedures and variables) can not be used while it is replaced. This restriction is necessary since there should be only one instance of the module's data at any time. We describe an implementation of our system that supports the blocking of references to a module's exported variables while the module is replaced in Chapter 6. After the modification and recompilation of a module, the old version can be replaced by the command

update <module id> **when** <proc list> **idle**

All the exported procedures of the module should be specified in <proc list> so that they are not used during the replacement. If any exported procedures are not specified, the command interpreter includes them in the when-part.

As for new modules, when a module is updated, its initialization code is executed. Instead of executing the initialization code, we may want to convert information stored in the old version to the new version. This conversion is discussed in a later section on "data restructuring".

4.3.3.2. Visible Modifications to a Module

Modifications to a module are visible if the recompilation of the new version makes other portions of the program inconsistent. Since the unchanged exported variables and procedures of a module are assigned the same addresses when the module is recompiled, the replacement of the module is visible only when the new version changes exported objects.

In most programming languages supporting separate compilation, such as Ada [47], Mesa [38], Modula-2 [51], if an exported object of a module is modified, other compilation units that use the module need to be (modified and) recompiled. This approach is not appropriate for dynamic modification of programs since the affected parts of compilation units that imported the changed object might be small portions of the compilation units. In our system, if an exported object is modified, only the affected parts of other modules that imported the object need to be (modified and) recompiled.

If the structure of an exported variable is changed, procedures that referenced (but not modules that imported) the exported variable need to be modified and recompiled to maintain the consistency of a program. After the modification and recompilation of such procedures, they are updated together with the module that exported the variable. If any of them are not included in the update command, the command interpreter prints out error messages. If their old versions are executed after the module is replaced, they may reference the changed variable as an old type. Therefore, the procedures that referenced the exported variable should be specified in a when-part to prevent such possibilities.

As an example of a changed exported variable, let us consider the program segment in Figure 4-7 and assume that procedures P and Q are the only procedures outside module M that use variable x. Suppose that module M is modified and variable x (but not t) is changed in the new version. When the new version is compiled, procedures P and Q become inconsistent since they use variable x; therefore, procedures P and Q are also modified and recompiled. After the modification and recompilation, module M and procedures P and Q can be replaced by the command

```
update M, P, Q when P, Q idle
```

```
module M;
  export x, t, S, T;
  (* x is a variable and t is a type *)
  (* S and T are procedures *)
  ...
end M
module N;
  export P, Q, R;
  import x, t;
  ...
  var v1, v2 : t;
  ...
  procedure P;  (* use x, t *)  end P;
  procedure Q;  (* use x *)  end Q;
  procedure R;  (* use v1, v2 *)  end R;
  ...
end N;
```

Figure 4-7. Outline of the module A and its uses.

Since the module's data representation is changed, the module is replaced while it is not in use as if procedures S and T are also specified in the when-part. Procedures P and Q are specified in the when-part to prevent their old versions from using variable x as an old type. If procedure P or Q is not updated with module M, the command interpreter generates an error message since the program will become inconsistent. Here a missing procedure can not be added to the update command by the command interpreter since it may not have been recompiled.

As with an exported variable, if the type of an exported procedure is changed, procedures that called (but not modules that imported) the exported procedure need to be modified and recompiled to maintain the consistency of a program. As before, such procedures are updated together with the module that exported the procedure. However, if the new version of the procedure contains a parameter convert routine, procedures that called the old version need not be recompiled or updated.

Unlike an exported variable or procedure, changes to an exported type can affect the data representation of modules that imported the exported type. The affected modules need to be modified, recompiled, and updated to remove inconsistency. However, to eliminate the unnecessary replacement of modules, our system reallocates the variables of the changed exported type when the module that exported the type is recompiled. Since such variables can be components of other variables, variables of imported types should be implemented using indirect address words. The use of indirect address words is necessary to prevent the reallocation of variables that contain imported type variables. We note that the reallocation of such variables can be done at compile-time or at load-time (versus at run-time) since there exists a fixed number of them. We believe that overhead of indirect address words is not a high price for the simplicity that we have gained.

Although modules need not be recompiled or updated, procedures that declared or referenced variables of the changed type are (modified and) recompiled since the structural details of the type have been changed. Such procedures are then updated with the module that exported the type. To prevent their old versions from referencing variables of the old type, they are specified in a when-part. Since the procedures are not executed when they are updated, there are no variables local in procedures of the type that need to be reallocated.

If an exported type is opaque; that is, its structural details are hidden outside the module, it is possible to implement the use of variables of the exported type independent of the structural details. Here procedures that declared and referenced such variables need not be recompiled. However, those procedures should be specified in a when-part instead of designing the dynamic modification system to reallocate variables of the exported type local to procedures. Although the reallocation is possible, it is expensive to implement (even if variables local to procedures are allocated in a heap as in the Mesa processor [28]). For example, to reallocate these local variables at run-time, their instances need to be located and then must be blocked from being referenced while they are reallocated.

As an example of a changed exported type, let us again consider the program segment in Figure 4-7. We assume that procedures P and R are the only procedures outside module M that use type t. Suppose that module M is modified and the structure of type t (but not x) is changed. When module M is recompiled, variables v1 and v2 are assigned new storage and the attributes of v1 and v2 in the symbol table data base are modified to reflect the change made to the exported type t. After module M has been recompiled, procedures P and R are modified and recompiled since procedure P uses type t and procedure R references variables v1 and v2. Then, module M and procedures P and R can be replaced by the command

update M, P, R **when** P, R **idle**

Procedures P and R are specified in the when-part to ensure that the uses of the old type t do not exist after the replacement.

Unlike an imported type, if the value of an imported constant is changed and if the change affects the data representation of a module, the whole module has to be recompiled and then updated. However, if the imported constant is used only within the procedure bodies of the module, only the procedures that used the constant need to be (modified and) recompiled and then updated.

4.4. Data Restructuring

To make a program easier to understand and to modify, Parnas advocates that the program should be divided into modules, where each module is the realization of some abstraction [40]. So we assume that a module implements an abstraction. There are two methods to define abstract objects using modules. Variables that represent an abstract object are declared within a module. Therefore, only one instance of an abstraction can exist. Alternatively, a module exports a type and an instance of that type is declared where an abstraction is used.

When a module is modified to use a different data representation, we may want to convert information stored in the old data representation into the new data representation. Data conversion should satisfy the following conditions:

(1) It should support an arbitrary conversion.

(2) It should allow the conversion to be carried out in a stable state that is specified by the programmer.

(3) It should not block the execution of other modified procedures and modules as long as they do not reference data that is being converted.

Since the initialization code of the new version can not reference the old data representation, we allow conversion routines to be defined with the new version. There are two classes of conversions: **local data conversion** and **exported type conversion**. The local data conversion is to initialize variables declared within the module. The exported type conversion is to initialize variables of an exported type that are declared outside the module.

4.4.1. Local Data Conversion

The new version of a module may contain a **local data convert routine** (called convert routine for short) that initializes the variables of the new version. If the new version contains the convert routine, the convert routine, instead of the initialization code, is executed when the module is updated; therefore, other necessary initialization steps should be duplicated in the convert routine. A syntactic description of a module with the convert routine is as follows:

```
<module decl> ::= [ interface ] module <module id>
                    [ <declarations> ]
                    [ <convert> ]
                    [ begin <stmt list> ]
                    end <module id>

    <convert> ::= convert
                    [ <export list> ]
                    [ <import list> ]
                    { <var decls> | <proc decls> }
                    [ before <stmt list> ]
                    [ after <stmt list> ]
                  end;

  <old var ref> ::= <old var id> |
                    <module id> . <old var id>
```

The scope of the convert routine of a module includes that of both the old and the new versions of the module. As before, references to an object of the old version can be qualified by the module name to resolve ambiguity.

The convert routine can export procedures that are defined in the convert routine or that are defined but not exported by the module to other convert routines. The import list specifies procedures that are used in the convert routine but not visible in the module. That is, the import list names procedures that are exported from other convert routines and that are visible in the enclosing scope of the module but not imported by the module. The export and import lists are necessary since the existing interface of a module may not be adequate to retrieve data stored in the module. For example, it is possible that the data stored in a module is meaningful only to users of the module. Here the stored data can be converted only from the convert routines of the users of the module. Therefore, if the necessary information can not be retrieved through the existing interface of the old version, the convert routine of the new version exports procedures that supply hidden details of the stored data.

As an example of the export and import lists, let us consider a module that provides two operations: StoreNew to store a new character string and IsIn to check to see if a given character string is already stored. Suppose character strings stored in the module are names and addresses. However, the module can never know whether a stored character string is a name or an address. We assume that another module keeps track of whether a stored character string is a name or an address. Suppose the module is changed to provide the StoreName, IsNameIn, StoreAddr, and IsAddrIn operations. Furthermore, when the module is replaced, the names and addresses stored in the old version should not be lost. Since the module can not distinguish between a name and an address, its convert routine can not divide the stored character strings into names and addresses. The division must be done by the convert routine of the module that remembered whether a stored character string is a name or an address. However, the stored character strings can not be retrieved through the StoreNew and IsIn procedures. Therefore, the convert routine needs to export a procedure, say NextElement, that returns the next character string stored in the old version whenever it is called. The convert routine of the other module imports the NextElement procedure and transfers the data stored in the old version to the new version using the NextElement, StoreName, and StoreAddr procedures.

When a module is updated, the before-part and then the after-part of the convert routine are executed. The module is unavailable for execution until the entire convert routine is executed. If other procedures and modules are also updated with the module, their new versions are available for execution as soon as they are updated. In particular, they are not blocked while the convert routine is executed. The procedures specified in the when-part can not be executed while the before-part of the convert routine is executed. Such procedures can be executed after the before-part has been executed; that is, the after-part and the specified procedures can be executed in parallel. Therefore, if the values of imported variables should not be changed while the convert routine is executed, such imported variables should be used within the before-part. Furthermore, procedures that can assign values to the imported variables need to be specified in the when-part of an update command to guarantee that their values are not changed. The after-part avoids unnecessary blocking of the procedures specified in the when-part during data conversion; especially, when a large data structure is converted.

If several modules that contain a convert routine are updated together, the procedures specified in the when-part are blocked until the before-part's of all the convert routines are executed. The convert routines are executed in the same order as their module names appeared in the update command. If a module contains nested modules, the convert routines of the nested modules are executed in the same order as their initialization code; that is, inner most ones first from top to bottom among the same nesting leveled ones. The nested modules are unavailable for execution until the convert routines of their enclosing modules are executed. The object code for convert routines is removed from the current system when the convert routines of all modules that are updated together are executed.

As an example of a local data convert routine, let us modify the BookKeeper module in Figure 1-2 to store customer names in one large common character array (a string space) instead of each name in a separate array. We assume that information stored in the old version should not be lost. Figure 4-8 outlines the new version of the BookKeeper module. Since no other parts of the program are affected, the change is transparent. When the BookKeeper module is updated, the initialization code of the NameStorage module is first executed. The convert routine of the BookKeeper module then transfers names stored in the old representation into the new representation. We note that if there were no ChangeIntoName procedure in the old version of the NameStorage module, the new version would

```
module BookKeeper;
  export StoreName, GetName, OpenAccount, AdjustBalance, GetBalance;
  import minAccountNo, maxAccountNo, string;

  module NameStorage;
    export ChangeIntoName, ChangeIntoString, nameType;
    import string;
    const maxNamePoolSize = 8000;
    type nameType = record start, length : integer end;
    var namePool : array 1 : maxNamePoolSize of char;
         availPtrNamePool : integer;

    (* The ChangeIntoName and ChangeIntoString procedures
      are changed to use the new data representation *)
  begin
     availPtrNamePool := 1;
  end NameStorage;

  var data : array minAccountNo : maxAccountNo of
              record
                name : nameType; balance : integer;
              end;
       availAccountNo : integer;

  (* The procedures OpenAccount, AdjustBalance, and
    GetBalance are not changed *)
  convert
    var i : integer; str : string;
    before
      availAccountNo := BookKeeper.availAccountNo;
      i := minAccountNo;
      while i < availAccountNo do
        NameStorage.ChangeIntoString(BookKeeper.data[i].name,str);
        ChangeIntoName (data[i].name, str);

      end;
  end;
begin
  (* This part is not changed *)
end BookKeeper;
```

Figure 4-8. Outline of BookKeeper with new data structures.

have contained a convert routine that exports such procedure for the convert routine of the BookKeeper module.

4.4.2. Exported Type Conversion

As we explained in the section on "Visible Modifications of a Module", if an exported type of a module is modified, the variables of the type declared outside the module are automatically reallocated. To initialize these variables, an exported type convert routine (called type convert routine for short) can be defined within the new version. The type convert routine is applied only to static variables outside the module since there are no active instances of variables of the type local to procedures when the module is updated. A syntactic description of the type convert routine is as follows:

```
<convert type> ::= convert <type id> (id1, id2);
                     { <var decls> | <proc decls> }
                     before <id list1> :
                          <stmt list>
                     after <id list2> :
                          <stmt list>
                   end;
```

As before, the scope of the type convert routine includes that of the old and new versions of the module containing the type convert routine. The type convert routine has two parameters whose types are implicit: one for a variable of the old type and another for a variable of the new type.

If a type convert routine is defined with the new version, only the variables specified in <id list1> and <id list2> are reallocated and then initialized by executing the type convert routine with the variables of the old and new types as parameters. As before, other procedures and modules that are updated with the module can be executed when the type convert routine is executed. The variables specified in the before-list are initialized by the statements of the before-part with the when-condition of the current update command maintained. The variables specified in the after-list are initialized by the statements of the after-part without blocking the procedures specified in the when-part; however, such variables are not available for use until they are initialized. We note that the conversion applied to the before-list need not be the same as that applied to the after-list. Furthermore, variables can be specified within both the before-list and the after-list.

If there are several type and local data convert routines in the new version, the convert routines are executed in the order of their appearance in the source code. The module becomes available for execution as soon as the before-part of the local data convert routine is executed. We note that type convert routines can be specified even when the module's data representation is not changed.

To minimize execution overhead for supporting type conversion, our system allows only one outstanding type conversion to variables at any time. That is, the new version of a module will not be updated if it contains a type convert routine for variables whose previous conversion has not been completed. We note that we could allow some fixed number of outstanding type conversions (with increased execution overhead); however, it is not clear that such a generalization is of practical interest.

As an example of an exported type convert routine, let us consider a module that implements small integer sets that can contain up to 100 integer values. Figure 4-9 outlines the module and its uses. We have assumed that there are three variables, s1, s2 and s3, of type smallIntSet and only two procedures, P and Q, use the module. Suppose that SmallIntSetModule is modified to support larger sets by increasing the value of maxSize from 100 to 200 (see Figure 4-10). After the recompilation of SmallIntSetModule, P and Q, the command

update SmallIntSetModule, P, Q **when** P, Q **idle**

replaces SmallIntSetModule, P, and Q. Procedures P and Q are specified in the when-part to prevent their old versions from using the new instances of s1, s2, and

```
module SmallIntSetModule;
  export smallIntSet, Insert, Remove, Has, Initialize;
  const maxSize = 100;
  type smallIntSet =
            record
              values : array 1:maxSize of integer;
              last : integer;
            end;
  procedure Insert (var s : smallIntSet; i : integer); ... end Insert;
  procedure Remove (var s : smallIntSet; i : integer); ... end Remove;
  procedure Has (s : smallIntSet; i : integer) : boolean; ... end Has;
  procedure Initialize (var s : smallIntSet); ... end Initialize;
end SmallIntSetModule;
var s1, s2, s3 : smallIntSet;
procedure P;
  ... Insert (s1, 10); Insert (s2, 5); Remove (s3, 27); ...
end P;
procedure Q;
  ... if Has (s1,5) then ...
end Q;
```

Figure 4-9. A small integer set module and its uses.

```
module SmallIntSetModule;
  export smallIntSet, Insert, Remove, Has, Initialize;
  const maxSize = 200;
  type smallIntSet =
           record
             values : array 1:maxSize of integer;
             last : integer;
           end;
  convert smallIntSet (old, new);
    procedure Copy (var newSet : smallIntSet;
        var oldSet : SmallIntSetModule.smallIntSet);
    begin
       for newSet.last := 1 to oldSet.last do
          newSet.values[newSet.last] := oldSet.values[newSet.last];
       end;
    end Copy;
    before s1, s2:
       Copy (new, old);
    after  s3:
       Copy (new, old);
  end;
  (* This part is not changed *)
end SmallIntSetModule;
```

Figure 4-10. Outline of new SmallIntSetModule.

s3 as the old type. Since variables s1 and s2 are specified in the before-list of the convert type routine (see Figure 4-10), the new instances of s1 and s2 are initialized by the Copy procedure when procedures P and Q are not executing. The new versions of procedures P and Q can be executed while the new instance of s3 is initialized; however, the execution of procedure P will be delayed if variable s3 is referenced before it is converted. We note that variables s1 and s2 could have been specified in the after-list.

We note that not all variables of exported types can be restructured through a type convert routine (see the NameStorage module in Figure 4-8 for an example). In general, a type convert routine can be used if information stored in a variable of the type is complete enough for its conversion. There are cases where both the local data and the exported type conversions have to be used.

4.4.3. Restructuring a Module Type

In other languages, such as Alphard [53], Concurrent Pascal [3], Euclid [31], and Simula [12], a module type can be defined and instances of the module type can be declared in other modules. If our language were extended to support module types, changes to a module type could be treated similarly to changes in an exported type. Furthermore, data stored in an instance of a module type could be converted with an exported type convert routine defined for the module type.

4.4.4. Comments and Alternative Approaches

Our scheme for data conversion supports the following three properties. First, information stored in a module's old data representation or variables of an old imported type are completely accessible for data conversion. This complete accessibility is necessary since the programmer may not foresee what information will be needed for data conversion. For example, if a module's current data representation is incorrect, information to be retrieved from the incorrect data structures depends on the unforeseen nature of a bug. Second, the programmer can ensure that data conversion is carried out in a consistent program state by using the when-part of an update command. That is, the correctness of data conversion does not have to depend on when (i.e., real time) the data conversion is carried out. Third, data conversion can be carried out in parallel with the execution of other modified procedures or modules as long as they do not reference data that are being converted.

Our approach to data conversion has a drawback. Since the exported type conversion has to be finished before the next type conversion, variables are converted even if they may never be referenced again. This drawback can be remedied by using version numbers and converting a variable when it is referenced as suggested by Fabry [18]. In this scheme, when a variable of an old representation is referenced, a chain of conversion routines is invoked to convert the variable into the current representation for that type.

A different data conversion scheme that uses a standard intermediate representation for each (exported) type is suggested in [23]. Herlihy and Liskov also use a similar scheme to communicate abstract values in messages in distributed environments [24]. (E.g., messages are transmitted using the standard representation.) The basic idea is that each implementation of a type be augmented with two operations, in and out [23] or encode and decode [24], that map values between the internal representation and the standard representation. If a type is implemented again, values stored in the current representation of the type can be converted into the new representation using the out (or encode) operation of the current representation and the in (or decode) operation of the new representation. The main advantage of this scheme is that the two operations can be provided when the type is implemented so that the programmer does not need to learn how the type is represented previously in order to construct a type conversion routine. Another advantage is that only one conversion routine is executed even if previous conversions have not been completed. The limitations of this scheme, when used for dynamic modification of programs, are that the definition (i.e., logical view) of a type can never be changed and that the in and out operations must be correct (which implies that each implementation of a type must be correct). Furthermore, every data type must define its standard representation and must provide two conversion routines, since every data type is a possible subject for dynamic modification.

4.5. Summary

We have described how procedures and modules can be added, deleted, and replaced from a running program. The desirable properties of our system are as follows: First, it allows only changes that maintain the consistency of a program. (We explain how this restriction is enforced by the command interpreter in Chapter 6.) Therefore, the source code matches the executable code in memory exactly. Since the old object code of procedures is not destroyed immediately, executable code in memory may not match the source code temporarily. However, the executable code will eventually match its source code. Second, the inconsistency of a program can be resolved by replacing only affected portions of the program. Third, any changes to a running program are possible as long as the resulting program is consistent. However, some changes may not be carried out if the when-condition of an update command is not satisfied. This happens if procedures specified in the when-part are used very frequently. In the next chapter, we explain how to find the smallest possible number of procedures and modules that need to be updated together to increase the success rate of an update command. Fourth, any information stored in the old version of a module can be converted into the new version. Finally, the programmer can define when changes should take place.

CHAPTER 5

METHODOLOGY FOR INCREMENTAL DYNAMIC CHANGES

5.1. Introduction

Suppose a set S of procedures and modules are modified and recompiled. Let us assume that S can be decomposed into $S_1,...,S_n$ such that the effect of updating S by a single command is equivalent to the effect of updating the S_i's as a sequence of commands.

Such a decomposition of S (that is, updating fewer procedures and modules at a time) is desirable for the following reasons. First, it helps the programmer to carry out changes to a running program in incremental steps. The incremental changes are preferred since the programmer needs to consider only a few procedures and modules at a time, which makes changing the program less complex.

Second, it reduces the contention with the running program since the user and dynamic modification processes will be competing for fewer procedures and modules that are to be replaced at a time. Furthermore, since procedures and modules are updated as an indivisible operation, they can not be executed while they are updated. Therefore, if fewer procedures and modules are updated by each command, fewer procedures and modules will be unavailable at a time.

Third, it might reduce the completion time to update S. The completion time includes the waiting time for the when-condition of an update command. In general, the when-condition will be satisfied more frequently when fewer procedures and modules are updated. For example, suppose we are adding a new feature to a compiler that consists of a scanner, parser and semantic routines. If all three components need to be changed for the new feature, they should be modified and updated one component at a time. Otherwise, it may be hard to find a moment when all three components are not in use.

In this chapter, we explain how S can be decomposed into a sequence of subsets. Our decomposition method is based on what programmers already know and do when they modify programs statically. That is, to modify a program, a programmer changes part of the program and then tests whether the program functions as expected with the partial change. This step is repeated until all changes to the program are made. Unlike static modifications, one can not afford to make mistakes in determining parts that can be modified separately when the program is changed dynamically. In particular, one has to ensure that the program will behave acceptably after each part is changed. So the decomposition is based on how to ensure that the program will behave acceptably. The meaning of an acceptable program behavior is also defined later.

This chapter is organized as follows: Section 5.2 explains our assumptions. Section 5.3 defines when a subset of S can be updated separately from the rest of S. Using the definition, the decomposition problem is precisely stated. Section 5.4 explains how to find an update precedence relation between two procedures and modules. The decomposition of S is based on the update precedence relations among procedures and modules in S. Section 5.5 describes an algorithm that finds a sequence of subsets that can be updated separately. We also explain when each subset can be updated. Section 5.6 discusses how each subset can be further partitioned. Section 5.7 shows how to decompose S when the program can satisfy some temporary functionality until all procedures and modules of S are updated. Section 5.8 uses the decomposition method for checking the consistency of a program after an update command. The last section summarizes the chapter.

5.2. Assumptions

We assume that there is a way to specify what programs, procedures, and modules do.

> **Definition:** We say that a program *satisfies* a specification if the program functions as described in the specification.

Similarly, we say that a procedure or module *satisfies* its specification if the procedure or module functions as described in the specification.

> **Definition:** We say that a procedure or module is *correct* if it satisfies its specification.

Formal specification and verification methods can be found in [41,25,53,16,5]. However, specifications and their verification can be informal for the purpose of this chapter if they can be used to answer the following kind of questions: Will procedure P satisfy its specification even if it calls the new version of procedure Q instead of the old version of procedure Q?

From now on, the letters, A and B, denote procedures or modules.

> **Definition:** We say that A *depends* on B if the correctness of B is necessary for the correctness of A.

That is, A depends on B if B needs to satisfy its specification for A to satisfy its specification. For example, A depends on B if A uses objects, such as procedures, variables, interrupt handlers, exception handlers, etc., defined in B.

In the remainder of this chapter, we use the following notations:

old(A) to denote the old version of A,

new(A) to denote the new version of A, and

P/A to denote a program P after A is updated.

To emphasize that the old version of A is replaced by the new version of A in the current program P, we use

old(A) / new(A)

to denote that the current program P is updated by A so that the old version of A is replaced by the new version of A. Here the current program P is implicit.

Definition: When A depends on B, we say that A *can use* new(B) *for* old(B) if A is still correct even if old(B) is replaced by new(B). Otherwise, we say that A *can not use* new(B) *for* old(B).

We denote A can use new(B) for old(B) by

A --> new(B) / old(B)

and A can not use new(B) for old(B) by

A -/-> new(B) / old(B).

We assume that the current program P satisfies an old specification OldSpec. That is, OldSpec is the description of what P does now. If the program contains bugs, OldSpec can be different from its required specification; that is, what it satisfies can be different from what it should satisfy. For procedures and modules, their old specifications refer to the descriptions of what they do now. Since the current program is running, we assume that its current specification OldSpec is acceptable, at least until all procedures and modules in S are updated.

Let NewSpec be a new specification that is to be satisfied by the program. For procedures and modules, their new specification refer to the descriptions of what they do when the program satisfies NewSpec. We assume that we are given a set S of procedures and modules that are to be modified to satisfy NewSpec. That is, the program will satisfy NewSpec after all procedures and modules in S are updated. We note that if a procedure or module depends on procedures and modules in S, its new specification can be different from the old specification even if it is not in S.

To make the decomposition of S meaningful, we assume that a given modification objective is time-independent. That is, the program will satisfy NewSpec whenever all the procedures and modules in S are updated regardless of when they are updated. This assumption is reasonable since the correctness of modification should not depend on when the program is changed. The assumption, however, does not mean that the procedures and modules in S can be updated in any order since the program should always function acceptably. (What constitutes an acceptable program is defined in the next paragraph.) We also assume that the programmer does not know which procedures are being executed when the program is updated.

Suppose the current program P is updated by a subset T of S.

> **Definition:** We say that P/T is *functionally consistent* if every procedure or module of P/T satisfies its old or new specification.

P/T represents the program P after update T has been applied. After T has been updated, it is possible that some procedures and modules depending on S satisfy their old specifications whereas the others satisfy their new specifications. Therefore, the definition is weaker than saying that P/T satisfies either OldSpec or NewSpec. For example, suppose a program handles two tape drivers and three terminals. Let us assume that the program is to be modified to support seven tape drivers and ten terminals. Suppose the program has been modified to handle seven tape drivers but still supports only three terminals. Clearly, this program satisfies neither the old specification nor the new specification. However, this program is functionally consistent if the following two conditions are true: First, there are no procedures whose old specifications depend on having exactly two tape drivers and three terminals. Second, there are no new procedures whose new specifications depend on having exactly seven tape drivers and ten terminals. Although a program is functionally consistent is weaker than saying it satisfies either the old or new specification, it is reasonable to assume that the functionally consistent program is acceptable while it is updated. So, for the current program P, we assume that the procedures and modules in S can updated in any order as long as the program is always functionally consistent.

5.3. Goals

To show how to decompose S into a sequence of subsets, we need to explain when a subset of S can be updated by itself.

> **Definition:** We say that a subset T of S can be updated *separately* (from S-T) to the current program P if P/T is functionally consistent.

This definition says that the program should always function consistently while the procedures and modules in S are updated. If a subset T of S can be updated separately from S-T to P, P/T/S-T, which is P after T and then S-T are updated, satisfies NewSpec (because of the time-independence assumption).

The above definition may seem overly restrictive since when a programmer changes a program, he may allow the program to satisfy some reduced functionality while it is modified. For example, the programmer may assume that tape drivers will not be used while the program is modified to handle another tape driver. We later discuss how to allow procedures and modules in S to satisfy a different (from OldSpec or NewSpec) specification while S is updated.

We now state the goal of this chapter as follows: for given CurSpec, NewSpec, and S, find a sequence $\{S_1,...,S_n\}$ such that $S = S_1 \cup ... \cup S_n$, S_i's are pair-wise disjoint, and each S_i, $1 \le i \le n$, can be updated separately from $S_{i+1} \cup ... \cup S_n$ to $P/S_1/S_2/.../S_{i-1}$. Update S_i will be applied to $P/S_1/S_2/.../S_{i-1}$ yielding $P/S_1/S_2/.../S_i$.

5.4. An Update Precedence Relation

To decompose S into a sequence of subsets that can be updated separately, we first define an update precedence relation on S by explaining when a procedure or module can be updated before or with another procedure or module. This update precedence relation defines equivalence classes on S. Procedures and modules in each equivalence class can be updated separately from other classes.

To preserve the functional consistency of the current program, whenever the old (or new) version of A is executed, it should satisfy its old (or new) specification. So we compute an update precedence relation between any two procedures and modules based on how to maintain the correctness of their old and new versions. Note that two procedures or modules, A and B, can be updated in the following three ways:

(1) update A before B;

(2) update B before A; or

(3) update A with B.

The order in which A and B are updated is significant only if it can affect the correctness of the old and new versions of A and B. In the next three sections, we explain when A and B need to be updated as (1), (2), or (3), respectively. To simplify our discussion, we introduce the following notations:

A < B if A should be updated before B.

A = B if A should be updated with B.

A <= B if A should be updated before or with B.

Furthermore, A will be enclosed by '[' and ']' if it should be updated when it is idle. For example,

[A] <= B

means that A should be updated before or with B when A is idle.

5.4.1. Correctness of the Old Version

Suppose A and B are to be replaced, where the old version of A depends on the old version of B. The new version of A may or may not depend on the new version of B; but it does not matter here. We explain how the correctness of the old version of A can be maintained.

If the old version of A can use the new version of B for the old version of B; that is, if

old(A) --> old(B)/new(B),

A and B can be updated in any order as far as the correctness of the old version of A is concerned. Suppose B is updated first and the old version of A is executed after B is replaced. Then, the old version of A still satisfies its old specification since the old version of A can use the new version of B for the old version of B. If A is replaced first, the correctness of the old version of A is still maintained since it can no longer be used.

As an example of a new version that can be used for an old version, consider the new AdjustBalance procedure in Figure 1-4. The new AdjustBalance procedure does what was expected from the old version and also checks for error conditions.

Although the source code of the ProcessRequest procedure in Figure 1-3 is not modified, its specification is changed to include the error checking for the Deposit and WithDraw transactions. Here the old specification of the ProcessRequest procedure can be satisfied using the new AdjustBalance procedure for the old AdjustBalance procedure.

If the old version of A can not use the new version of B for the old version of B; that is, if

old(A) -/-> old(B)/new(B),

the old version of A should not be executed after B is replaced. This restriction implies that A should be updated before or with B. Furthermore, the old version of A should not be in use when B is updated. Since we assumed that programmers do not know which procedures are being executed, the programmer has to specify a when-condition to assure that the old version of A is not used when B is updated. So A can be updated when A is idle and then B can be updated as follows:

U1. **update** A **when** A **idle; update** B

or A can be updated and then B can be updated when A is idle as follows:

U2. **update** A; **update** B **when** A **idle**

However, the update sequence U1 is preferred for the following three reasons.

First, the first command of U1 requires only the old version of A to be idle when A is updated. The second command of U2 requires both the old and the new versions of A to be idle when B is updated. The condition "**when** A **idle**" as used in U1 is simpler to check and is probably satisfied more frequently. Second, if A is never idle, B will not be updated in U2. Here the changes to A may need to be undone. However, since A is not replaced in U1, no changes need to be undone. Third, requiring A to be specified in a when-part when it is updated is simpler to remember especially if S is large.

In summary, to preserve the correctness of the old version of A, if

old(A) -/-> old(B)/new(B),

then

[A] <= B.

As an example of how to maintain the correctness of the old version, suppose two procedures P and Q are to be replaced, where the old version of procedure P calls the old version of procedure Q. Let us assume that the old version of P can not call the new version of Q instead of the old version of Q; that is,

old(P) -/-> old(Q)/new(Q).

If procedure Q is replaced before procedure P and procedure P is executed, then the old version of procedure P may call the new version of procedure Q. Furthermore, if the old version of procedure P is being executed when procedure Q is replaced, the old version of procedure P may also call the new version of procedure Q. Therefore, to prevent the old version of P from ever calling the new version of Q, P should be updated before or with Q when P is not executed. That is, procedures P and Q can be updated as follows:

update P **when** P **idle; update** Q.

We note that they can also be updated together when procedure P is idle.

As another example, suppose procedure P and module M are to be replaced, where procedure P references a variable exported from module M. Let us assume that the new version of module M has changed the type of the exported variable referenced within procedure P. Here procedure P should be updated before or with module M when procedure P is idle to prevent procedure P from referencing the exported variable as the old type.

As a special case for the correctness maintenance of the old version, suppose procedure P depends on procedure P; that is, procedure P calls itself. If the old version can not call the new version, procedure P should be updated before or with procedure P when procedure P is idle; that is, procedure P should be updated when it is idle.

5.4.2. Correctness of the New Version

Suppose A and B are to be replaced, where the new version of A depends on the new version of B. If

new(A) --> new(B)/old(B),

A and B can be updated in any order as far as the correctness of the new version of A is concerned. For example, let us consider two procedures P and Q, where the new version of procedure P calls the new version of procedure Q. Suppose the new version of procedure Q improves the old version of procedure Q by employing a better algorithm but the old and new versions satisfy the same specification. Here the new version of procedure P can use the old version of procedure Q. That is, the new version of procedure P can satisfy its specification using the new version of procedure Q instead of the old version of procedure Q. So procedures P and Q can be updated in any order as far as the correctness of the new version of procedure P is concerned.

Suppose the new version of A can not use the old version of B for the new version of B; that is,

new(A) -/-> new(B)/old(B).

Here, the situation that the new version of A might use the old version of B should be avoided. That is, B should be updated before or with A for the new version of A to be correct. We note that A need not be idle when A is replaced since we are only concerned about the correctness of the new version of A.

In summary, to ensure the correctness of the new version of A, if

new(A) -/-> new(B)/old(B),

then

B <= A.

For example, suppose two procedures P and Q are to be replaced, where the new version of procedure P calls the new version of procedure Q. Let us assume that the new version of procedure P can not use the old version of Q for the new version of Q; that is,

new(P) -/-> new(Q)/old(Q).

If procedure P is replaced and the new version of procedure P is executed before procedure Q is updated, the new version of procedure P may call the old version of procedure Q. To prevent such a call, procedure Q should be updated before or with procedure P. That is, they can be updated as follows:

update Q; **update** P.

As before, they can also be updated together.

5.4.3. The Update Together Relation

We define an update together relation, =, on S by saying that A and B in S are related if A <= B and B <= A. That is, if

A <= B and B <= A,

then

A = B.

It is easy to show that this relation is an equivalence relation that partitions S into equivalence classes such that procedures and modules in each class should be updated together.

As an example of an update together relation, consider the changes made to the GetBalance and PrintAccount procedures in Figure 4-1. Since the parameters of the old GetBalance is different from those of the new version,

old(PrintAccount) -/-> old(GetBalance)/new(GetBalance).

Thus,

[PrintAccount] <= GetBalance.

Furthermore, since

new(PrintAccount) -/-> new(GetBalance)/old(GetBalance),

the PrintAccount procedure can not be updated first; that is,

GetBalance <= PrintAccount.

Therefore,

[PrintAccount] = GetBalance.

That is, they should be updated together when the PrintAccount procedure is idle as shown at line (7) in Figure 4-1.

5.4.4. Domain of Update Precedence Relations

We say that A directly depends on B if A depends on B and the definition of B is needed to compile A. In finding update precedence relations, we need to consider A and B only if A directly depends on B (or vice versa) because of Lemma 1 and Lemma 2.

Lemma 1: Suppose A depends on C because A depends on B and B depends on C, but A does not directly depend on C. If A <= C then A <= B and B <= C.

Proof: Suppose A <= C. That is, C < A is false. That is, old(A) -/-> old(C)/new(C). Since the old version of A uses C through B, this means that old(A) -/-> old(B)/new(B) and old(B) -/-> old(C)/new(C). Therefore, A <= B and B <= C. Q.E.D.

Lemma 2: Suppose A depends on C because A depends on B and B depends on C, but A does not directly depend on C. If C <= A then C <= B and B <= A.

Proof: Suppose C <= A. That is, A < C is false. That is, new(A) -/-> new(C)/old(C). Since the new version of A uses C through B, this means that new(A) -/-> new(B)/old(B) and new(B) -/-> new(C)/old(C). Therefore, B <= A and C <= B. Q.E.D.

For example, suppose procedure P calls procedure Q and procedure Q calls procedure R, but procedure P does not call procedure R. Then, we need not find update precedence relations between procedures P and R, it follows from update precedence relations between procedures P and Q and between procedures Q and R.

5.5. An Update Sequence

This section explains how to decompose S into a sequence of subsets and how to find procedures that need to be idle when a subset is updated. The idea behind the decomposition of S can be described as follows: When a program is to be changed by S, the new versions of some procedures and modules of S can be used for their old versions. Similarly, the old versions of some procedures and modules of S can be used for their new versions. Such procedures and modules form subsets of S that can be updated separately.

When a subset is updated, procedures outside the subset may need to be specified in the when-part to ensure that the program does not function unexpectedly. Such procedures need to be specified because they use procedures or modules of the subset but can not use the old versions and the new versions at the same time.

5.5.1. The Update Dependency Graph

To decompose S into a sequence $S_1,..,S_n$, we build an update dependency graph defined as follows:

> **Definition:** The *update dependency graph* (UDG) of S is a directed graph (N,E). N is a set of procedures and modules. A directed edge (A,B) is in E if B <= A.

Because of Lemma1 and Lemma2, edges between A and B are computed only if A directly depends on B (or vice versa). From now on, we use "depend" to mean "directly depend". We note that when edges of UDG are computed, each node of N is marked as whether it should be idle when it is updated; that is, whether it should be specified in the when-part of an update command. N includes the following procedures and modules:

(1) Procedures and modules that are deleted or modified; that is, S.

(2) Procedures whose specifications are changed even if they are not modified or recompiled.

(3) Procedures whose specifications and code are unchanged but directly use procedures and modules of the first two kinds.

The procedures and modules of the first kind are updated or deleted. The procedures of the second and third kinds are included in UDG for two reasons. First, they may need to be specified in a when-part; however, they never need to be updated since their object code is not changed. Second, they may be needed to determine update precedence relations between two procedures and modules that do not directly depend on each other. We note that the UDG of S can be constructed in time proportional to the number of "depend on" relations among procedures and modules in (1), (2), and (3).

For example, suppose procedure P calls procedures Q and R. Let us assume that only procedures Q and R are modified. Although procedure P is not modified, it is possible that procedure P may not be able to use the old (or new) version of procedure Q and the new (or old) version of procedure R. That is, procedure P can use either the old versions of procedures Q and R or the new versions of procedures Q and R. So procedures Q and R should be updated together when procedure P is idle. Such relation can be derived from update precedence relations between procedures P and Q and between procedures Q and R. Here, we have the following subgraph:

$$\begin{tikzcd} & {[P]} \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] & \\ Q & & R \end{tikzcd}$$

update Q, R **when** P **idle**

We note that this sort of subgraph is possible even if the specification of procedure P is not changed.

As another example, suppose procedure P calls procedure Q and procedure Q calls procedure R. Let us assume that only procedures P and R are modified and recompiled. It is possible that procedures P and R need to be updated together. Such relation can be found by considering update precedence relations between procedures P and Q and between procedures Q and R. That is, we may have the following subgraph:

$$\begin{tikzcd}
{[P]} \arrow[d, leftrightarrow] \\
{[Q]} \arrow[d, leftrightarrow] \\
R
\end{tikzcd}$$

update P, R **when** P, Q **idle**

This subgraph means that procedures P and R should be updated together when procedures P and Q are idle as shown in the above figure.

5.5.2. Finding an Update Sequence

We now explain how to find an update sequence from the UDG of S.

Definition: Let G = (N,E) be a directed graph. We can partition N into equivalence classes Ni, $1 \leq i \leq n$, such that nodes v and w are equivalent if and only if there is a path from v to w and a path from w to v. Let E_i be the set of edges connecting the pairs of nodes in N_i. The graphs $G_i = (N_i,E_i)$, $1 \leq i \leq n$, are called the *strongly connected components* of G.

A strongly connected component of UDG defines a subset of S that can be updated separately. A strongly connected component with no dangling out edges can be updated first. There is a well-known algorithm that finds all strongly connected components of a directed graph in time proportional to max(|N|,|E|) [1]. Using that algorithm, we can find the strongly connected components $\{E_i\}$, $1 \leq i \leq n$, of UDG.

Definition: The *reduced UDG* of S is a directed graph (N,E), where each node n_i represents a strongly connected component E_i of UDG and (n_i,n_j) is in E if there are v in E_i and w in E_j such that (v,w) is an edge in the UDG of S.

We note that the reduced UDG of S contains no cycles (this follows from the definition of the strongly connected component).

As an example of a reduced UDG, let us assume that we have the following UDG with N equals $\{P_1,\ldots,P_{11}\}$.

$$\begin{tikzcd}
 & {[P_1]} \arrow[d, leftrightarrow] & & & & & \\
 & P_2 \arrow[dl] \arrow[d, leftrightarrow] \arrow[dr, dash] & & {[P_3]} \arrow[dl] \arrow[dr] & & {[P_4]} \arrow[d] \arrow[dr, leftrightarrow] & \\
P_5 \arrow[r] & {[P_6]} & P_7 \arrow[r] & P_8 \arrow[u] & P_9 \arrow[r, leftrightarrow] & P_{10} & P_{11}
\end{tikzcd}$$

The SCC's of this UDG and the reduced UDG are as follows:

SCC's:

$E_1 = \{P_9, P_{10}\}$

$E_2 = \{[P_3], P_7, P_8\}$

$E_3 = \{[P_1], P_2, P_5, [P_6]\}$

$E_4 = \{[P_4], P_{11}\}$

reduced UDG:

$$\begin{tikzcd}
E_1 \arrow[d] & & E_4 \arrow[ddl] \\
E_2 \arrow[dr] & & \\
 & E_1 &
\end{tikzcd}$$

The reduced UDG defines update precedence relations among the SCC's of UDG. Since a node of the reduced UDG with no dangling out edges can be updated first, we can find an update sequence $\{S_1,\ldots,S_n\}$ from the reduced UDG as follows:

```
"Let {E_i} be the strongly connected components of UDG";
"Initialize G = (N,E) to the reduced UDG";
i := 1;
while N not empty do
   "Find a node nj with no out edges";
   S_i := E_j; i := i + 1;
   E := E - {all edges coming into n_j};
   N := N - {n_j};
end
```

A sequence is built by repeatedly determining a SCC that can be updated next assuming that other SCC's that need to be updated before the SCC are already updated. The update sequence is not unique since there can be several nodes of G to choose from the fifth statement in the algorithm; that is, the reduce UDG defines a partial ordering. The algorithm terminates since the reduced UDG has no cycles. In the previous example, three possible update sequences are as follows:

(1) E_1, E_2, E_3, E_4
(2) E_1, E_2, E_4, E_3
(3) E_1, E_4, E_2, E_3

After E_1 is updated, E_2 or E_4 can be updated next since either subset can be updated before the remaining subsets.

5.5.3. When to Update a Subset

When a subset of S is updated, the marked procedures of the subset should be specified in the when-part of an update command. However, if fewer procedures are specified in the when-part, the condition when they are not executed can probably be checked with less run-time overhead. For example, the condition may need to be checked whenever a process returns from a procedure specified in the when-part. So it is desirable to specify as few procedures as possible in the when-part. The procedures that can only be invoked (directly or indirectly) from other procedures specified in the when-part can be eliminated since such procedures are not executed whenever the other procedures are not executed. We note that this elimination of procedures from a when-part does not make the condition to be satisfied sooner.

After each subset is updated, the old versions of updated procedures and modules that are executing still satisfy their old specifications. Furthermore, other procedures and modules that are not updated satisfy their old specifications since if they were not, they would have been already updated. Finally, the new versions of the updated procedures and modules satisfy their new specifications whenever they are executed. Therefore, the program is functionally consistent after each subset is updated.

5.6. A Better Update Sequence

In decomposing S into subsets, it is desirable to have as many subsets as possible. If there are too many subsets, we can always merge some of them into a single subset. In this section, we explain how S might be decomposed into more subsets.

Update precedence relations found by considering only two procedures or modules at a time are conservative. That is, we may find that A should be updated before or with B (or vice versa) when it need not be so. We extend the "can use for" notation defined in Section 5.2 to allow several procedures and modules as follows:

A --> E,F,G / B,C,D

denotes that A can use B, C, and D for E, F, and G, respectively.

To show why the previous update precedence relation is conservative, suppose we have procedures P, Q, R, and S, where procedure P calls procedure Q or procedure R depending on a result from a call to procedure S. That is, we have

```
procedure P;
  ...
  if S then Q else R end;
  ...
```

Let a new procedure Q' equal procedure Q, a new procedure R' equal procedure R, and a new procedure S' return the inverse value of procedure S. Then,

old(P) -/-> S/S'.

Thus,

P <= S.

However,

old(P) --> R,Q,S / Q',R',S'

that is, procedure P can collectively use procedures Q', R', and S' for procedures R, Q, and S, respectively. That is, procedure P modified as follows:

```
procedure P;
  ...
  if S' then Q' else R' end;
  ...
```

satisfies the old specification of procedure P. Here procedure P need not be updated before or with procedure S if procedures Q, R, and S are updated together when procedure P is idle.

Similarly, although

new(P) -/-> S'/S,

it is possible that

new(P) --> R',Q',S' / Q,R,S.

Here, procedures Q, R, and S need not be updated before or with procedure P if they are updated together.

In general, suppose a procedure or module A depends on several procedures and modules within UDG. Let us assume that, for a subset, say B, C, and D, of such procedures and modules, we have

old(A) -/-> old(B)/new(B),

old(A) -/-> old(C)/new(C), and/or

old(A) -/-> old(D)/new(D)

However,

old(A) --> old(B),old(C),old(D) / new(B),new(C),new(D)

Then, the edges from B, C, and/or D to A can be removed from UDG. That is, A need not be updated before or with B, C, and D. Here, new edges need to be added among B, C, and D to ensure that they are updated together.

$$\begin{tikzcd}
 & A & & & & A & \\
B \arrow[ur] & C \arrow[u] & D \arrow[ul] & & B \arrow[r, leftrightarrow] & C \arrow[r, leftrightarrow] & D
\end{tikzcd}$$

That is, the subgraph on the left side can be transformed into the subgraph on the right side.

Similarly, if the new version of A can collectively use the old versions of B, C, and D for their new versions; that is, if

new(A) --> new(B),new(C),new(D)/old(B),old(C),old(D),

the edges from A to B, C, and D can be removed from UDG. Again, new edges need to be added among B, C, and D to ensure that they are updated together.

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[d] \arrow[dr] & & & & A & \\
B & C & D & & B \arrow[r, leftrightarrow] & C \arrow[r, leftrightarrow] & D
\end{tikzcd}$$

That is, the subgraph on the left side can be transformed into the subgraph on the right side.

Based on the above edge transformations, the UDG of S can be modified to contain more SCC's as follows:

```
"Let G = (N,E) be the UDG of S";
"Let M be a set of nodes that depend on more than one node in N";
for each n in M do
  if there is an edge-transformation that
     increases the number of SCC's then
       "Apply the edge-transformation to G";
  end;
end;
```

This algorithm might not give an UDG that contains the maximum number of SCC's because of the following reasons: When dependent procedures and modules are tested for edge transformation, all different combinations of them should be tried to

see which edge transformation would result in more SCC's. Furthermore, it is possible to get a different number of SCC's depending on the order that edge transformations are applied. Therefore, to find the maximum number of SCC's, edge transformations have be applied in all possible orders (which requires an exponential time). However, we note that the condition for adding and deleting edges of UDG described in this section will be satisfied rarely in most programs. Furthermore, even when the condition is satisfied, adding and deleting edges as described may not increase the number of SCC's. Therefore, the order of edge transformations is not significant in most programs. That is, we believe that the algorithm will almost always result in an UDG that contains the maximum SCC's for most programs.

5.7. Reduced Functionality

We have explained how S can be partitioned into $\{T_1,...,T_n\}$ such that the program satisfies the old or new specification after each T_i, $1 \leq i \leq m$, is updated. However, it may be possible to allow the program to satisfy a different specification while the program is modified. Suppose we know that the program does not have to satisfy the old or new specification while S is updated. Then, the restriction that the program should always satisfy either the old or new specification after each subset of S is updated can be relaxed. For example, if a program operates a tape driver, we may be able to assume that the tape driver will not be used while the program is modified to handle another tape driver.

Suppose there is a temporary specification TempSpec such that the program can satisfy TempSpec while it is modified. Then, S can be decomposed into a sequence of subsets using TempSpec instead of OldSpec. We note that if a program is assumed to satisfy a different specification, specifications for procedures and modules also change. So the new versions of different procedures and modules can be used for their old versions. That is, we may be able to partition S into different subsets. In particular, if the temporary specification describes a reduced functionality, the new versions of more procedures and modules can be used for their old versions. Therefore, S can be partitioned into more subsets.

As an example of reduced functionality, suppose the on-line banking system is modified to store up to two names for each account. Furthermore, the Open transaction is also modified to request for an optional name. If we know that the Open transaction will not be requested while the system is modified, the modification can be carried out in two parts. The first part includes the BookKeeper module and PrintAccount procedure. The second part includes the OpenAccount procedure. The first part can be further divided using the method described in the previous sections; the decomposition depends on how they are modified. For example, if the new BookKeeper module exports two new procedures Get2ndName and Put2ndName without changing the existing GetName and PutName procedures, the BookKeeper module can be updated first. Then, the new PrintAccount procedure can be updated. Otherwise, they may need to be updated together. After the BookKeeper module and PrintAccount procedure have been updated, the OpenAccount procedure can be updated to request two names.

5.8. The Consistency Checker

In this section, we explain how a Compilation Dependency Graph (CDG), which is a subgraph of UDG, can be built when procedures and modules are recompiled or deleted. CDG is used by the consistency checker, which is part of the command interpreter, to check whether a program will be consistent after the current update command. Unlike for finding an update sequence, there are no restrictions on programs to apply consistency checking. We say that a procedure or module of a program is *inconsistent* if the procedure or module can not be correctly compiled.

For example, if a procedure references an array variable as record, the procedure is inconsistent.

> **Definition:** The *compilation dependency graph* (CDG) of the current program is a graph G = (N,E). N is a set of procedures and modules that are recompiled or deleted. Furthermore, N also includes procedures and modules that have become inconsistent because of the recompilation or deletion of other procedures and modules in N. An edge (v,w) is in E means that the recompilation of v has made w inconsistent or vice versa.

CDG, after each edge (v,w) substituted by two directed edges (v,w) and (w,v), is a subgraph of UDG since the consistency maintenance of a program is a special case of the correctness maintenance of a program. Unlike UDG, an edge (A,B) in CDG means that A and B should be updated together. For example, if A depends on B and the new version of B causes the old version of A to become inconsistent, then the old version of A can not use the new version of B. Furthermore, the new version of A that is created to confirm to the new version of B can not use the old version of B. Therefore, A and B should be updated together when A is idle.

When a procedure or module is recompiled, other procedures and modules that used the old version of the recompiled procedure or module can become inconsistent. Such affected procedures and modules can be determined at compile-time. (The next chapter explains how they are determined.) So we assume that there is an affected function, named Af, that returns a set of procedures and modules that become inconsistent when a given procedure or module, A, is recompiled; that is,

Af(A) = {procedures and modules that become inconsistent because of the recompilation of A}

The program can also become inconsistent when a procedure or module is deleted.

If a procedure is deleted, other procedures that called the deleted procedure become inconsistent. If a module is deleted, other procedures and modules that used the deleted module become inconsistent. Such affected procedures and modules can be determined using identifier cross reference lists, which are described in the next chapter. So we assume a use function, named Uf, that returns the set of procedures and modules that use a given procedure or module, A; that is,

$$Uf(A) = \{\text{procedures and modules that use } A\}$$

CDG is initially empty; it is expanded as procedures and modules are recompiled or deleted. Each node of CDG is marked as whether it has been recompiled and as whether it has been deleted. Figure 5-1 shows how CDG is modified when a procedure or module is recompiled or deleted. For simplicity, we assume that procedures and modules are not recompiled more than once without

```
procedure AddAffected (n : node; S : setOfNodes);
begin
   for each m in S do
      if m not in N then "Add node m to N"; end;
      "Add an edge (n,m) to E";
   end;
end AddAffected;

compile A:
   if A not in N then "Add node A to N" end;
   "Mark A as recompiled";
   AddAffected (A, Af(A));

delete A:
   if A not in N then "Add node A to N" end;
   "Mark A as deleted";
   AddAffected (A, Uf(A));
```

Figure 5-1. The compile and delete operations on CDG.

being updated first. Whenever a procedure or module, A, is recompiled or deleted, it is added to CDG (if not already added) and marked accordingly. If A needs to be added, A is added as a new root in the forest. Procedures and modules that have become inconsistent because of the recompilation or deletion of A are added to CDG. The edges from A to them are also added to CDG.

CDG usually consists of several connected components. A connected component of CDG defines procedures and modules that need to be updated together to maintain the consistency of the program. When an update command is requested, the delete operation on CDG is applied for each procedure or module whose definition is not deleted but whose name is specified in the delete-part of the updated command. Such delete operations are necessary since the definitions of procedures and modules need not be deleted prior to the update command. The command interpreter ensures the consistency of a program by checking that if a node is specified in the update or delete part of the update command, then all other nodes within the connected component of CDG that contained the node are also specified in the update or delete part. Furthermore, all the nodes specified in the update or delete part should have been marked as recompiled or deleted. Otherwise, the command interpreter generates an error message saying that the current update command is inconsistent to the programmer and does not carry out the current update command.

5.9. Summary

We have described how to incrementally modify a program with the restriction that the program always satisfy its old or new specification. Furthermore, we have shown how to relax the restriction to allow the program to satisfy a temporary specification while it is modified. We believe that the decomposition method developed can help the programmer to carry out program modification in a step-wise manner. That is, the programmer first finds a partition and then modifies and updates each part at a time in a sequence. As a special case, we have shown how the consistency of a program after an update command can be ensured.

CHAPTER 6

IMPLEMENTATION CONSIDERATIONS

6.1. Introduction

Having discussed a programming system that supports the dynamic modification of a program and developed a methodology for incremental dynamic changes, we now discuss how to implement our system. The discussion is based on our experience with implementing the prototype system.

In the next section, we explain how the command interpreter is implemented and how a procedure or module is recompiled with complete type-checking. We also describe a program structure tree that is used to maintain source code and symbol tables.

To support the efficient dynamic modification of a program, a run-time program representation is proposed. Section 6.3 describes the layout of executable code in memory. Furthermore, we extend addressing modes and procedure call and return instructions to provide the necessary built-in synchronization and mutual exclusion capability.

To modify executable code in memory dynamically, a process, called the dynamic modification process, is executed in parallel with other user processes. Section 6.4 explains how the dynamic modification process changes the executable code in memory to incorporate new object code and data. To provide a block of memory for new object code and data, the existing code and data may need to be relocated. Section 6.5 describes how the code and data can be relocated at run-time without halting program execution and without creating dangling pointers.

6.2. Implementation Details

Our system allows the recompilation of any procedures and modules with complete type-checking. This compilation strategy can be used for a system that supports the development and maintenance of programs. Rudmik and Moore describe a program development system that is based on a similar compilation strategy [43]. However, their scheme requires that the skeleton of a program structure be compiled first to define procedures and modules that can be compiled separately. Our system derives the structure of a program during compilation. Furthermore, any procedures and modules can be added and deleted.

In this section, we provide a general view on how our system works by describing the implementation of the command interpreter. We also describe a symbol table organization that supports the recompilation of a procedure and module with type-checking. Furthermore, we explain how CDG is updated and how the conversion routines described in Chapter 4 are compiled.

6.2.1. The Program Structure Tree

The backbone of our system is a program structure tree (PST) that represents hierarchical relations among modules and procedures. The structural information is used, first, to check the validity of arguments to the edit, compile, delete, and update commands; second, to retrieve the source code of a requested procedure and module; and third, to reconstruct a compile-time environment for a procedure or module. DeRemer and Kron use a similar program structure tree to describe module interconnectivity information for large programs [15]. Their program structure tree is to support "programming-in-the-large" whereas our program structure tree is to support "programming-in-the-small"; therefore, different information is stored in a node.

The program structure tree of a program is a directed tree (N,E) with a root r. Each node of {N - r} represents a module or procedure of the program. The root node, r, is a virtual node in the sense that it does not correspond to any procedure or module. A directed edge (v,w) in E means that procedure or module w is nested within procedure or module v. The set E also includes edges from the root node to initially separately compiled modules.

A subtree of PST is created by the compiler when a module or procedure is compiled. Each node contains the following attributes:

(1) type of a node;

(2) current source code;

(3) provisional source code;

(4) object code (and data if the node is a module);

(5) symbol table;

(6) export list;

(7) import list.

The type of a node is a procedure, process, or module. In the remainder of this chapter, a procedure refers to either a procedure or a process. Furthermore, we may use a "node" to refer to a procedure or module corresponding to the node. We assume that names of procedures and modules are all distinct. The current source code will always corresponds to executable code in memory. Provisional source code is created when a procedure or module is modified or created. The object code from the last compilation is stored in a node. The symbol table contains the definitions of objects declared in a node. The export list contains, for each exported identifier, procedures and modules that referenced the identifier. It is used to determine the affected procedures and modules to maintain CDG when a node is recompiled. The import list contains identifiers that are used within the node but declared outside the node. It is used to update the export lists of other nodes when the node is recompiled. Although our programming language does not allow the export and import lists to be specified within a procedure, the export and import lists are also defined for procedure nodes.

6.2.1.1. Symbol Table Organization

Symbol tables are organized to support the recompilation of an arbitrary procedure or module with complete type-checking. A stack of symbol tables is used during compilation. Whenever a scope is opened for a procedure or module, a new symbol table is pushed on the symbol table stack. When the scope is processed, a symbol table is popped and saved in the procedure or module's node. The saved symbol table is later used to compile (new) procedures and modules within the scope.

In our language, a procedure defines an open scope and a module defines a closed scope. The open scope means that a name in the enclosing scope is visible within the current scope unless it is redeclared. The closed scope means that no names (except pervasive identifiers such as the names of library routines) of the enclosing scope are visible within the current scope. The import list of a module explicitly specifies names of the enclosing scope that are to be visible within the module. A symbol table for a closed scope contains all user defined names visible within the scope. Each symbol table is marked as closed or open to control the inheritance of names from enclosing scopes. To look up an identifier during compilation, an instance of the identifier is searched in the symbol table on top of the stack. If no such entry is found, the search is repeated in next symbol tables until the correct entry is found or a symbol table of a closed scope is encountered, beyond which the search can not continue.

We note that instead of a separate symbol table for each procedure or module, one symbol table can be used to implement several symbol tables to reduce space overhead. Cook and LeBlanc describe such a symbol table organization [11]. Each procedure or module is assigned a unique identification number, termed lexical level number (LLN). A symbol table stack is replaced by a stack that contains the lexical level numbers of enclosing scopes. Since each identifier instance is uniquely identified by a (name, LLN) pair, an instance of an identifier is searched with its name and LLN on top of the stack. If no entry is found, the search is repeated with the next LLN on the stack.

6.2.1.2. The Export and Import Lists

If a node represents a module, the *export* list of the node contains cross-reference (called *used*) lists for constants, types, and variables that are exported and defined in the module. Within PST, there can be only one used list for each identifier. In particular, if an exported identifier is declared within a nested module, a used list for the identifier is contained in the export list of the nested module only. A used list for an identifier contains procedures and modules that can become inconsistent if the type of the identifier is changed. Since procedures and modules nested within a module are also recompiled whenever the module is recompiled, such procedures and modules can not become inconsistent. So they are not included in the used lists of the export list for the module.

A used list of an exported variable contains procedures (but not modules) that can become inconsistent when the type of the variable is changed. The used list does not contain any modules since the inconsistency can be resolved by modifying and recompiling the affected procedures only. A used list of an exported constant contains procedures and modules that can become inconsistent if the value of the constant is changed. A module is included in the used list if the constant is referenced to define the data representation of the module; otherwise, only procedures that referenced the constant are included. A used list of an exported type contains procedures and modules that can become inconsistent if the structure of the type is changed. Furthermore, for each module included in the used list, the used list also contains the variables of the exported type. This variable list is for the reallocation of such variables when the exported type is modified.

Since a procedure can not export any objects, the export list of a procedure node contains only one used list that is for the procedure itself. This used list contains other procedures that called the procedure. As before, only procedures that are declared outside the procedure are included in the list.

The **import** list of a node contains procedures that are declared outside the node but are called within the executable part of the node. The executable part of a node means the initialization statements or the procedure body. The import list also contains (identifier, module) pairs for identifiers that are declared outside the node but are referenced within the data representation and executable parts. If a node is a procedure, the import list need not contain (identifier, module) pairs for identifiers that are defined within the enclosing module since the procedure is recompiled whenever the enclosing module is.

6.2.2. The Command Interpreter

As we said in Section 3.4.1, the command interpreter accepts a request from the programmer and then starts an appropriate component of the system. Figure 6-1 outlines the implementation of the command interpreter. When a command is requested, the command interpreter checks the validity of arguments; that is, each argument should name a unique procedure or module. For the simplicity of our discussion, we assume that all recompiled procedures and modules are eventually updated. That is, it is not necessary to be able to undo the effect of compilation on symbol tables, CDG, and PST. Otherwise, changes to symbol tables, CDG, and PST have to be delayed until procedures and modules are successfully updated.

```
loop
  "Wait for a command";
  "Check the validity of arguments";
  case command of
    Edit :
      begin
        "Get the source code of an argument";
        "Start the editor with the source code";
        "Wait until the editing session is completed";
        "Store the new provisional source code";
      end;
    Compile :
      begin
        "Get the source code of an argument";
        "Start the compiler with the source code";
        "Wait until the compilation is completed";
        "Update CDG and PST";
      end;
    Delete :
      begin
        "Delete the symbol table of an argument";
        "Update CDG and PST";
      end;
    Update :
      begin
        "Check the consistency of a program";
        if not consistent then
            "Print error messages";
            exit case;
        end;
        "Start the dynamic modification process";
        "Wait for completion";
        if succeeded then
            "Update CDG";
        else
            "Print error messages";
        end;
      end;
  end;
end;
```

Figure 6-1. Outline of the command interpreter.

When an edit command is requested, if there is a procedure or module corresponding to an argument, the editor is started with either the current or the provisional source code as discussed in Section 3.4.3. If the argument names a new procedure or module, the editor is started with empty source code. Here a new node is created in PST for the new procedure or module. After the editing session is completed, the new provisional source code is stored in the node of the argument. We note that the provisional source code is not yet decomposed into a subtree.

When a compile command is requested, the compiler is started with the provisional source code for an argument. The compiler generates a new subtree whose root is the argument. The new subtree replaces the corresponding old subtree of PST. After the compilation, the export lists of nodes outside the subtree are modified using the import lists of the nodes in the old and new subtrees. For example, if a new node references an identifier defined in a module but the old node did not reference the identifier, the new node is added to a used list for the identifier in the export list of the module. Conversely, if an old node referenced an identifier defined in a module but the new node did not reference the identifier, the old node is deleted from a used list for the identifier in the export list of the module.

The export list of a new node is created from the export list of the corresponding old node. The new export list inherits old used lists for identifiers whose types are not changed. If the type of an identifier in the old export list is changed, CDG is modified to include the procedures and modules specified in a used list for the identifier. For example, if the type of an exported variable is changed, procedures that are specified in a used list for the variable in the old export list become inconsistent. Therefore, such procedures are added to CDG. If an identifier in the old export list is not exported from the new node, the type of the identifier is considered to be changed.

For a delete command, the definitions of the arguments are deleted from symbol tables. Then, all procedures and modules specified the used lists in the export lists of the arguments are added to CDG. Finally, nodes corresponding to the arguments are deleted from PST.

When an update command is requested, the consistency of a program after the update command is checked using CDG (as described in Section 5.8). If the program is consistent, the object code of the arguments are linked to resolve external references. The dynamic modification process is then started to load the new object code into memory. If the new object code are loaded successfully, the updated procedures and modules are deleted from CDG and their provisional source code become their current source code. As we will see later, the dynamic modification process can fail if there is not enough available space in memory or if the when-condition of an update command is not satisfied within some fixed time limit.

6.2.3. Recompilation of Procedures and Modules

When a procedure or a module is recompiled, the symbol table stack is reconstructed to contain the symbol tables of enclosing scopes for type-checking. Since procedures and modules can not reference identifiers visible within enclosing scopes that are beyond the first closed scope, the symbol table stack contains only the symbol tables of enclosing scopes up to the first closed scope.

When a procedure is compiled, a new symbol table is created to contain parameter and local variable definitions. This symbol table is marked as an open scope. The used list of procedures and imported (by the enclosing module) constants, types, and variables is built during the compilation. After the procedure is compiled, an entry that contains the definition of the procedure is inserted in the symbol table of the enclosing scope. An entry for the old version is removed from the symbol table if the procedure is recompiled.

If the new version of a procedure contains a parameter conversion routine, the conversion routine is compiled using the old and new symbol tables for the procedure. Since "var" parameters are considered as pointer variables within a parameter conversion routine, the "var" parameter entries in the old and new symbol tables are changed to denote them as pointers. Furthermore, the addresses of the parameters of the new version are changed to point to a correct location within a run-time stack since when the convert routine is executed, the current activation record is that of the old (not new) version. After a new procedure is compiled, the changed entries in the new symbol table are restored to compile the procedure body.

If the new version of a procedure contains a labeled statement, the beginning location of a statement that matches the label within the old procedure code segment in memory is computed by recompiling the old version. We note that the source code of the old version is the current source code stored in the procedure node. When the procedure is updated, a jump instruction is inserted in that location of the old procedure code segment by the dynamic modification process.

When a module is compiled, if it exports an identifier, an entry for the identifier is inserted in the symbol table of the enclosing scope. The entry is filled with compile-time attributes when the identifier is declared. If an identifier is imported, the attributes of an entry for the identifier in the enclosing scope's symbol table are copied into the entry in the current symbol table. The original entry of an exported identifier (that is, the entry in the symbol table of the module that declared the identifier) keeps a cross reference list of symbol tables that contain copy entries. This cross reference list is used to update the copy entries when the original entry is modified. We note that a cross reference list for an identifier is usually different from a used list for the identifier in the export list of a module that declared the identifier.

If a module is recompiled, exported variables whose types are not changed should be allocated the same addresses within the module's data space. To support this address assignment scheme, the symbol table for the old version is searched for exported variables to create an available data space list before starting the recompilation. If a variable is exported from the old and new versions and its type is not changed, it is allocated at the same location. Otherwise, the variable is allocated at a new location using the available data space list. If a variable is exported from the old version but is not assigned the same space, its old space is added to the available data space list. A similar scheme is used to assign the same addresses to the unchanged exported procedures of a module.

If the new version of a module contains a local data conversion routine or exported type conversion routines, they are compiled using the symbol tables of the old and new versions.

6.3. The Architecture

Our architecture (that is, run-time program representation, addressing modes, and procedure call and return instructions) was designed to support the dynamic modification of programs. The design goals of the architecture were (1) to support the efficient implementation of dynamic modification of procedures and modules; (2) to support data conversion during the replacement of modules; (3) to permit the relocation and recovery of code and data segments; and (4) to work on a multiprocessor system with shared memory.

The architecture is based on a stack (however, an architecture need not be based on a stack to support dynamic modification). We describe only features that are necessary for dynamic modification and explain why such features are necessary. For example, built-in data types and operators, sizes of a memory word and memory , I/O instructions, etc., are not described.

6.3.1. Run-Time Representation of a Program

As mentioned above, a primary goal of the architecture is to represent a program so that procedures and modules can be replaced efficiently. A program is represented by using two types of address tables during execution as in Lilith [52]. However, each module is assigned a unique module number even if it is nested within another module. A module number is used as an index in the module address table (MAT) that contains addresses of the modules' procedure address tables and data frames (see Figure 6-2). The procedure address table (PAT) of a module contains the beginning addresses of procedures defined in the module. A data frame is a contiguous area of memory allocated to the variables of a given module.

The format of an address entry in MAT and PAT is as follows:

```
+---+---------------------+
| l |       address       |          l = available/locked
+---+---------------------+
```

The physical address is derived by appending zero to the address stored in an entry. We note that addresses can be stored using one less bit by placing PAT, data frames, and code segments at even addresses. The lock bit 'l' is used to enforce mutual exclusion among processes referencing or modifying an address entry.

A procedure's code is placed in a segment and the beginning address of the segment is stored in the procedure's address entry in PAT. Since no information in code depends on its location, the code can be placed anywhere in memory and can be relocated at run-time. The first word at the beginning of each procedure's code segment is reserved for as an awaited bit and a use count; that is, the first word is

```
    Module                Procedure
Address Table           Address Table
+------------+<-PB +-->+------------+<-CB +-->+-+----------+
|            |     |   |            |     |   |w| use count|
+-+----------+     |   |            |     |   +-+----------+
|1|    ------+-----+   +-+----------+     |   |            |
+-+----------+         |1|    ------+-----+   | procedure  |
|1|    ------+---+     +-+----------+         |   code     |
+-+----------+   |     |            |     PC->|            |
|            |   |     |            |         +------------+
|            |   |     +------------+
+------------+   |
                 |       Module
                 |     Data Frame
                 +---->+------------+<-DB
                       |            |
                       |            |
   Process             |            |
    Stack              +------------+
+------------+<-LP
|            |
|            |
+------------+<-SP  Registers:
| procedure  |        CB  pointer to the current PAT
|   locals   |        CM  current module number
+------------+        DB  pointer to the current module's
|   saved    |            data space
| registers  |        FP  pointer to the current activation
+------------+<-FP        record
|   actual   |        HP  pointer to the high end of a stack
| arguments  |        LP  pointer to the low end of a stack
+------------+        PB  pointer to the module address table
|   return   |        PC  pointer to the current instruction
|    value   |        PN  current process identifier
+------------+        RL  header of the process ready list
|            |        SP  pointer to the top of a stack
+------------+<-HP    TC  total use count register
| process    |
|descriptor  |
+------------+<------------+     +-------> +------------+
                           |     |         | process    -+-->...
                     +-+--+--+--+--+       |descriptor  |
                     |1|  |  |  |  |       +------------+
                     +-+-----+-----+
                     Ready List (RL)
```

Figure 6-2. A program's representation at run-time.

viewed as follows:

```
+---+----------------------+
| w |      use count       |
+---+----------------------+
```

w = awaited bit
use count = no. of active calls

The awaited bit is set to one if the procedure is specified in the when-part of the current update command. The use count records the number of active calls to the procedure. The use count is allocated within the procedure's code segment because if several versions of the procedure coexist, the use count of each version should keep the number of active calls to that version. Separate use counts are necessary to determine when each version can be garbage collected.

Variables local to a procedure are allocated within the stack as shown in Figure 6-2. The variables of a module are allocated within the module's data frame. Although the variables of a module are allocated within the module's data frame, the data space for variables of an imported type are allocated within the data frame of the module that exported the type. For example, let us consider the module declarations in Figure 6-3. Since the types of variables y and z of module N are imported from modules M and N1, respectively, the data space for variables y and z are allocated within the data frames of modules M and N1, respectively. Because data space for variable y is allocated in the data frame of module M, if module M is replaced and the size of type t1 is increased, the data frame of module N need not be relocated. As with an address entry in MAT and PAT, one bit of indirect address words is used as a lock bit. This lock bit is checked to see whether exported type conversion is pending before variables of an imported type are referenced.

A set of registers contain frequently used values during instruction execution. Register PB (Program Base) points to the beginning of the module address table. Registers CB (Code Base) and DB (Data Base) point to the procedure address table

```
module M;               Data Frame for     Data Frame for
   export t1;             Module N           Module M
   ...                  +-----------+      +-----------+
                        |    x      |      |    own    |
end M;                  +-+---------+      | variables |
module N;               |1|  y  ----+----->+-----------+
   import t1;           +-+---------+      |           |
   module N1;           |1|  z  ----+--+   +-----------+
      export t2         +-+---------+  |   |           |
      ...               |           |  |   +-----------+
                        |           |  |
   end N1;              +-----------+  |   Data Frame for
   var                                 |     Module N1
      x : integer;                     |   +-----------+
      y : t1; z : t2;                  |   |    own    |
   ...                                 |   | variables |
                                       |   +-----------+
end N;                                 |   |           |
                                       +-->+-----------+
                                           |           |
                                           +-----------+
```

Figure 6-3. Storage allocation for imported type variables.

and data frame, respectively, of a module that is currently executed. The current module number is stored in register CM. Registers LP (Low end Pointer) and HP (High end Pointer) point to the two ends of the stack for the current process. The current process identification number is stored in register PN (Process Number). Register SP (Stack top Pointer) points to the top of the stack. Register FP (Frame Pointer) points to the activation record of a procedure that is being executed.

To support multiprogramming, the ready list register (RL) is provided. The lock bit l of register RL is to enforce mutual exclusion among processes modifying the ready list. The other two fields (head and tail) point to the queue of processes that are ready to execute. Each process descriptor contains a copy of the values of all processor registers defining the environment of the process.

Register TC keeps the total use count of the procedures specified in the when-part of the current update command. As with register RL, register TC contains a lock bit to enforce mutual exclusion among processes modifying the register.

6.3.2. Locks

To provide mutual exclusion among processes referencing and modifying a memory word, the following Lock and Unlock operations are provided:

```
Lock (adr):
  loop
    "Exclude processes";
    if Mem[adr].l = 0 then exit loop;
    "Release exclusion";
  end;
  Mem[adr].l := 1;
  "Release exclusion";

Unlock (adr):
  Mem[adr].l := 0;
```

"Mem[adr].l" denotes the lock bit of a memory word at address adr. With the Lock operation, a lock bit is tested and set as an indivisible operation. If the lock bit is already set, the if-statement within the loop-statement is retried after the exclusive control over the memory word has been released and then regained. The Unlock operation clears the lock bit of a memory word so that processes can lock the memory word again. A Lock and Unlock pair is intended to encapsulate a critical section of instructions.

6.3.3. Addressing Modes

The addressing scheme for operands is designed to support the relocation of data frames and stacks at run-time without creating dangling pointers and the conversion of a module's data representation and variables of an exported type. To support the above two goals, lock bits and logical addresses are used during execution. A logical address is represented by a module number and an offset, where the module number zero is reserved for stacks. The logical address is translated into a physical address only when it is needed; for example, to reference an object and to store an object into memory.

Instructions can be divided into four basic categories: load and store instructions, operators, control, and miscellaneous instructions. The operators and control instructions, except for call and return instructions, are not described in this thesis; a complete discussion on the operators and control instructions can be found in [7].

The load and store instructions transfer data between a stack or data frame and the top of the stack, where they are accessed by operators. The load and store instructions require a single operand since the stack address is implicit. An operand represents a logical address and is in one of the following seven modes: a local, global, external, indirect, stack, index, or immediate mode. The logical address is translated into a physical address for load and store instructions. However, load address instructions push a logical address onto the stack. The pushed logical address is translated into a physical address when the physical address is needed to reference and to store data. This delayed translation is necessary to allow the relocation of data frames and stacks without creating dangling pointers for variables and addresses stored in stacks.

To show how logical and physical addresses are computed and used during execution, we explain how the load (LOAD) and load address (LOADA) instructions are executed with different modes (data types are ignored). We note that registers PB, CB, DB, FP, SP, HP, and LP contain physical addresses.

Local mode: Variables local to a procedure are specified by an offset n. Instructions LOAD and LOADA in the local mode are executed as follows:

```
LOAD local n
  inc (SP); Stack[SP] := Mem[FP + n];

LOADA local n
  inc (SP); Stack[SP] := (0, FP+n-LP);
```

"(0, FP+n-LP)" represents a logical address with the module number 0 and the offset FP+n-LP. The module number zero in the logical address denotes that the offset is relative to register LP. The logical address of a local variable is pushed onto the stack since the stack might have been relocated when the pushed address is used.

Global mode: A variable of the current module is specified by an offset n. Instructions LOADA and LOAD in the global mode are executed as follows:

```
LOADA global n
  inc (SP); Stack[SP] := (CM,n);

LOAD global n
  Lock (PB + CM*2);
  inc (SP); Stack[SP] := Mem[DB + n];
  Unlock (PB + CM*2)
```

The logical address is (CM,n), where register CM contains the current module number. "PB + CM*2" refers to the address of the current module's data frame entry in MAT. The Lock and Unlock operations are needed to mutually exclude the relocation and the use of the module's data frame. We note that if the data frame of a module can be relocated only when the module is not being executed (that is, when all of the module's exported procedures are not executed), the Lock and Unlock pair is not necessary for the LOAD instruction in the global mode.

External mode: An imported variable is referenced by its module number m and offset n. Instructions LOADA and LOAD in the external mode are executed as follows:

```
LOADA external m,n
  inc (SP); Stack[SP] := (m,n);

LOAD external m,n
  Lock (PB + m*2);
  adr := Mem[PB + m*2]; inc (adr, n);
  inc (SP); Stack[SP] := Mem[adr];
  Unlock (PB + m*2)
```

The module number is used to fetch the beginning address of the module's data frame in MAT. We note that locking and address fetching can be carried out with only two memory references to MAT. Again, the Lock and Unlock operations provide the necessary mutual exclusion.

We note that if the LOAD instruction is executed indivisibly and if one processor is shared among all processes of the current program, the Lock and Unlock pair can be replaced by a simpler operation, Spin, that is defined as follows:

```
Spin (adr):
  while Mem[adr].l = 1 do "Process switch" end;
```

That is, the LOAD instruction is executed as follows:

```
LOAD external m,n
  Spin (PB + m*2);
  adr := Mem[PB + m*2]; inc (adr, n);
  inc (SP); Stack[SP] := Mem[adr];
```

The Spin operation checks whether the data frame of module m can be accessed. Here, checking is sufficient since we have assumed that the data frame can not be locked while the LOAD instruction is executed.

Indirect mode: A variable of an imported type is addressed with the indirect mode by an offset or by a module number and an offset depending on whether the variable is declared within the current module. Since an indirect word for such variable contains the logical address of data space for the variable, the LOADA instruction in the indirect global mode is executed as follows:

```
LOADA indirect global n
  loop
    Lock (PB + CM*2);
    if Mem[DB + n].l = 0 then exit loop end;
    Unlock (PB + CM*2)
  end;
  inc (SP); Stack[SP] := Mem[DB + n];
  Unlock (PB + CM*2);
```

As before, the Lock and Unlock operations provide the necessary mutual exclusion. To ensure that an exported type conversion to the variable is not being carried out or pending, the lock bit of the indirect word is checked. As we described in Section 4.3.3, if an exported type of a module is changed, procedures that use the exported type are not executed when the module is updated. That is, the lock bit can not be set while the instruction is executed; therefore, the lock bit is only checked (vs. locked). The LOADA instruction in the indirect external mode is executed similarly using m instead of CM.

The LOAD instruction in the indirect global mode is executed as follows:

```
LOAD indirect global n
  loop
    Lock (PB + CM*2);
    if Mem[DB + n].l = 0 then exit loop end;
    Unlock (PB + CM*2)
  end;
  (m1,n1) := Mem[DB + n];
  Unlock (PB + CM*2);
  Lock (PB + m1*2);
  adr := Mem[PB + m1*2]; inc (adr, n1);
  inc (SP); Stack[SP] := Mem[adr];
  Unlock (PB + m1*2);
```

"(m1,n1)" is the logical address of the data space for the variable. This logical address is translated into a physical address to load data onto top of the stack from memory. As before, the Lock and Unlock pairs provide the necessary mutual exclusions. The LOAD instruction in the indirect external mode is executed similarly using m instead of CM.

As with the external mode, if we assume that the LOAD and LOADA instructions are executed indivisibly and that there is only one processor in the system, a Lock and Unlock pair can be replaced by a Spin operation. For example, the LOAD instruction in the indirect global mode can be executed as follows:

```
LOAD indirect global n
  loop
    Spin (PB + CM*2);
    if Mem[DB + n].l = 0 then exit loop end;
    "Process switch";
  end;
  (m1,n1) := Mem[DB + n];
  Spin (PB + m1*2);
  adr := Mem[PB + m1*2]; inc (adr, n1);
  inc (SP); Stack[SP] := Mem[adr];
```

The indirect local mode is also supported for "var" parameters. When a procedure with a "var" parameter is called, the logical address of an actual argument to the "var" parameter is pushed on the stack. Then, the stored logical address is referenced by the indirect local mode within the procedure body.

Stack mode: Instructions in the stack mode do not contain an explicit operand; top of the stack contains the logical address of the operand. The logical address is translated into a physical address as for an operand in the external mode. If the module number of the logical address is zero, the LP register is used as the base address (see the above LOADA instruction in local mode). The LOADA instruction is not defined for this mode.

Index mode: This mode is for accessing an array element. The two top elements of the stack and an instruction's parameter are used in a new address calculation. The computed index on top of the stack is multiplied by the parameter that is the size of the accessed data type. The result is then added to the offset of the logical address stored on the stack to form a new logical address. The physical address is computed from the new logical address for the LOAD instruction in the index mode.

Immediate mode: This mode is used to generate constants. The parameter is the value to be loaded on the stack.

6.3.4. Procedure Call Instructions

The procedure call instructions provide the necessary synchronization and mutual exclusion capability to implement the update command. There are two instructions: one for calling local procedures (that is, those defined within the same module) and another for calling external procedures (that is, those defined in other modules).

Figure 6-4 defines the local call instruction. The address of a procedure address entry is computed from the CB register and a given procedure number. There are two ways that the execution of a procedure can be delayed. The Lock operation on "p_entry" can be delayed if it is already locked; that is, either another process is executing a call instruction to the same procedure or the procedure is being replaced. Alternatively, the current process may loop temporarily within the call instruction if its awaited bit is set and the TC register equals zero. The awaited bit is set if the procedure is specified in the when-part of the current update command. The TC register keeps the total use count of the procedures specified in the when-part of the current update command. If the TC register equals zero, the when-condition of the current update command is satisfied; that is, the dynamic modification is in progress. Therefore, the procedure can not be executed. The beginning address of the procedure body is recomputed within the loop-statement since the procedure's code segment might have been replaced.

If a procedure can be executed, the use count of the procedure's code segment is incremented with the procedure's address entry locked to guarantee

```
call local procedure p:

  registers m_entry, p_entry, p_adr;
  Save Registers CB, CM, DB, PC, FP;
  loop
     p_entry := CB + p;
     Lock (p_entry);
     p_adr := Mem[p_entry];
     if Mem[p_adr].w = 0 then
        (* Modification to the procedure is not pending *)
        (* Increment the use count *)
        inc (Mem[p_adr]); Unlock (p_entry);
        exit loop;
     else
        (* Modification to the procedure is pending *)
        Lock (TC)
        if TC ≠ 0 then
           (* OK to execute the procedure *)
           (* Increment the total and use count *)
           inc (TC); Unlock (TC);
           inc (Mem[p_adr]); Unlock (p_entry);
           exit loop;
        end;
        Unlock (TC); Unlock (p_entry);
     end;
     (* Retry since modification is currently being done *)
  end;
  Save Register p_adr;
  FP := SP; PC := p_adr + 1;
```

Figure 6-4. Local procedure call instruction.

that it is modified or used by one process at a time. If the procedure is specified in the when-part of the current update command and if the TC register is not equal to zero, the procedure can be executed. Here the TC register is incremented to maintain the correct total use count of the procedures specified in the when-part. The TC register is also locked during the increment to prevent it from being used and changed simultaneously. The use count of the procedure's code segment is still incremented even when its awaited bit is set because the current update effort may be aborted if the when-condition is not satisfied within some fixed time limit.

The beginning address of the procedure's code segment (i.e., p_adr) is saved so that the use count of the same procedure code segment can be decremented when a process returns from the procedure. This save is necessary since the procedure address entry can be changed to contain the beginning address of a new procedure code segment while the old code segment is executed.

Figure 6-5 defines the external procedure call instruction. An external procedure is called with a module number and a procedure number. The differences between the local and external call instructions are that the latter instruction, first, requires two parameters: a module and a procedure numbers; second, assigns new values to the CM and DB registers; third, checks that the module is available before

```
call external procedure p defined in module m:

 registers m_entry, p_entry, p_adr;
 Save Registers CB, CM, DB, PC, FP;
 CM := m;
 m_entry := PB + CM*2 - 1;
 loop
    Lock (m_entry);
    CB := Mem[m_entry];
    DB := Mem[m_entry+1];
    p_entry := CB + p * (size of an entry in bytes);
    Lock (p_entry);
    p_adr := Mem[p_entry];
    Unlock (m_entry);
    if Mem[p_adr].w = 0 then
       ...
       (* This part is the same as that in Figure 6-4 *)
       ...
    end;
    (* Retry since modification is currently being done *)
 end;
 Save Register p_adr;
 FP := SP; PC := p_adr + 1;
```

Figure 6-5. External procedure call instruction.

using PAT; and fourth, recomputes the beginning address of PAT and stores in the CB register within the loop-statement since PAT could have been relocated.

Figure 6-6 defines the procedure return instruction. The procedure return instruction decrements the use count of the code segment which is pointed to by p_adr. As we said before, the code segment pointed to by p_adr can be different from that pointed to by the procedure address entry if the procedure is replaced while executed. If the awaited bit is set, the TC register is decremented and checked to see whether the pending modification can be started. The pending modification is started by sending the updateOK signal to the dynamic modification process when the total use count becomes zero.

To support the relocation of PAT's at run-time, the ChangeCB instruction is defined as follows:

```
ChangeCB (m,adr):
  if CM = m then CB := adr end
```

```
registers p_entry, p_adr;
p_entry := CB + p;
Restore Registers  CB, CM, DB, PC, FP, p_adr;
Lock (p_entry);
dec (Mem[p_adr]);   (* decrement the use count *)
if Mem[p_adr].w = 0 then
   (* Modification to the procedure is not pending *)
   Unlock (p_entry);
else
   Unlock (p_entry);
   Lock (TC); dec (TC);
   (* Signal if pending modification can be started *)
   if TC = 0 then send (updateOK) end;
   Unlock (TC);
end;
```

Figure 6-6. Return instruction.

that is, the CB register is assigned a new address if the module m is the current module in execution. The instruction is executed as an indivisible operation. If we do not have the ChangeCB instruction, the local call instruction should be modified to compute the address of the current module's PAT within the loop-statement as was done for the external call instruction since PAT might be relocated.

6.3.5. Monitor Enter and Leave Instructions

The same call instruction is used for both an interface procedure and a regular procedure to make conversion from a regular module to an interface module (or vice versa) transparent to the users of the module. However, to enforce mutual exclusion among processes that want to enter an interface module, an interface procedure body is encapsulated by the monitor Enter and Leave instructions.

Each interface module is allocated an interface word that has the following format:

```
+---+-------+-------+------+-------+
| l | owner | count | head | tail  |
+---+-------+-------+------+-------+
```

All the fields of an interface word must be initialized to zero. The lock bit "l" is used to enforce mutual exclusion among processes referencing and modifying an interface word. If an interface module is free, the owner field of its interface word contains zero; otherwise, it contains the identifier of the process that is currently executing within the module. The count field keeps the number of times the owner process has called but not yet returned from procedures of the interface module. The head and tail fields are used to maintain the queue for the processes waiting to enter the interface module. We have taken a simple view of monitors and have not addressed many of the problems associated with their implementation, for example, scheduling among waiting processes, release of exclusion while waiting inside of a monitor after nested monitor calls, etc. A complete treatment of the implementation details of monitors can be found in [10].

The first instruction in the body of a procedure of an interface module is the Enter instruction defined in Figure 6-7. If the interface module is free, this instruction sets the owner field to the current process. If the procedure is called

```
Enter (adr):
  (* adr is the address of an interface word *)
  Lock (adr);
  if Mem[adr].owner = 0 then
     (* Interface module is free *)
     Mem[adr].owner := PN;
     Unlock (adr)
  elsif Mem[adr].owner = PN then
     (* Nested call to the interface module *)
     inc (Mem[adr].count);
     Unlock (adr)
  else
     (* Current process can't enter the module *)
     "Put the current process into the waiting queue"
     Unlock (adr)
     "Resume a process in the ready list"
  end;

Leave (adr):
  Lock (adr);
  if Mem[adr].count = 0 then
     (* Current process is leaving the module for good *)
     if Mem[adr].head = 0 then
        (* No waiting processes for the module *)
        Mem[adr].owner := 0;
     else
        "Remove a process from the waiting queue
         and set the process as the owner of the module"
        "Put the process into the ready list"
     end;
  else
     (* Current process has nested calls *)
     dec (Mem[adr].count);
  end;
  Unlock (adr);
```

Figure 6-7. Monitor Enter and Leave instructions.

by the owner process, it increments the count. Otherwise, the current process is put into the waiting queue of the interface module. Upon the completion of the procedure, the Leave Instruction in Figure 6-7 is executed before the procedure return instruction. The Leave instruction decrements the count if the owner process is still active within the interface module. Otherwise, if processes are not waiting for the module, the interface module is marked as free. If processes are waiting, a waiting process is removed from the queue and the removed process becomes the owner of the module. To reduce the execution time of the Enter and Leave instructions, the count corresponds to one less than the actual number of calls to interface procedures.

6.4. The Dynamic Modification Process

The dynamic modification process modifies the current core image to incorporate the new object code and data of updated procedures and modules. The process is executed in parallel with other user processes; therefore, we need to ensure that an inconsistent version of a program is not executed while the program is updated. That is, the following three conditions should be satisfied by an implementation of the dynamic modification process. First, when several procedures are updated together, the dynamic modification process should update them as an indivisible operation. This restriction is necessary since if a process executes a new procedure and if it calls another updated procedure, the new version of the called procedure should be used. Second, processes should be prevented from using inconsistent data structures while data is converted. Third, the dynamic modification process should be able to recognize when the when-condition of an update command is satisfied and should be able to sustain the condition as long as it is needed. In this section, we explain our implementation of the dynamic modification process that satisfies these conditions. We note that there can be at most one outstanding update request at any time.

6.4.1. The Linker and Loader

When a module or a procedure is compiled, its object code contains symbolic references to external variables and procedures. When an update command is requested, the linker converts the symbolic external references and calls to module numbers and procedure numbers. The loader copies the new object code and data into memory. Since the newly loaded code and data are not accessible from the running program until entries in MAT and PAT's are updated, the loading step does not require any synchronization or mutual exclusion with user processes.

6.4.2. Code Layout for Procedures

Because of the use of procedure address tables, replacing a procedure is simple and efficient after the new code has been loaded; only one procedure address entry has to be changed (See Figure 6-8). However, replacing several procedures and modules requires elaborate coordination between the dynamic

```
              Procedure
            Address Table
            +------------+        +----->+------------+
            |            |        |      |    old     |
 procedure  +------------+       ///     | procedure  |
   entry    |       -----+--------+      |    code    |
            +------------+        |      +------------+
            |            |        |
            +------------+        +----->+------------+
                                         |    new     |
                                         | procedure  |
                                         |    code    |
                                         +------------+
```

Figure 6-8. After the replacement of a procedure.

modification and user processes as we will see in the next sections.

Figure 6-9 shows a code layout after a procedure with a parameter convert routine has been updated. A new address entry points to the new code segment. The old address entry is made to point to the code segment of the parameter convert routine that creates and initializes actual arguments to the new version and that calls the new version. Note that calls to the old version (but not the new version) will go through the parameter convert routine.

6.4.3. Updating Procedures

When several procedures are updated together, each procedure's address entry is first locked to prevent processes from starting the procedure during the modification. After all the procedures have been locked, their address entries are modified to point to the new object code. Processes waiting for the procedures to

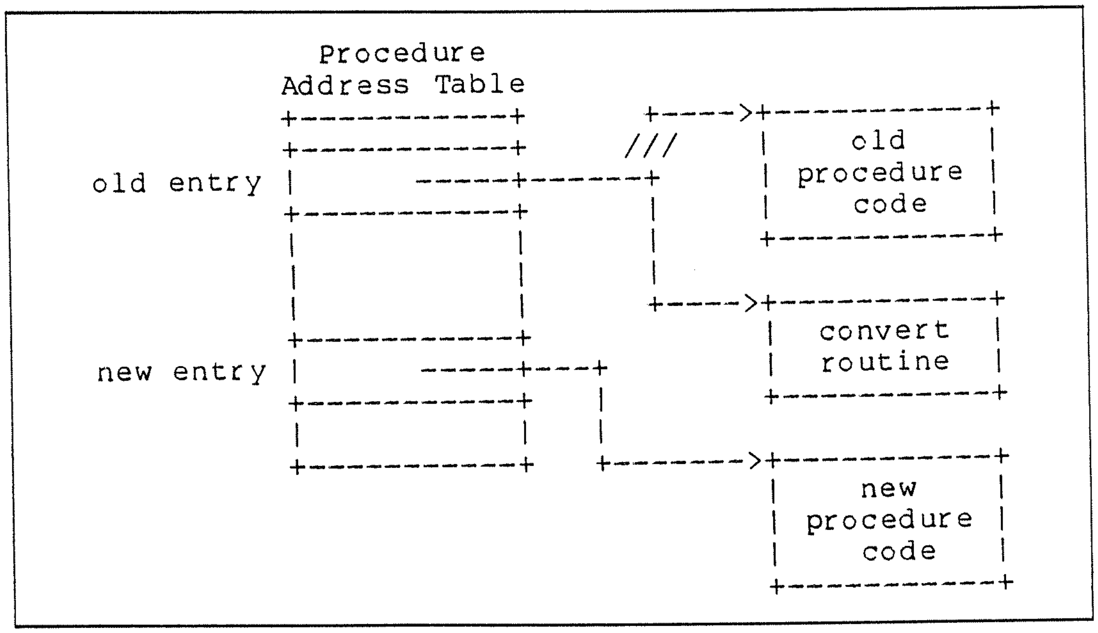

Figure 6-9. After a procedure with a parameter convert routine has been updated.

become available can resume their execution when the address entries are unlocked. That is, the command

$$\text{update } P_1,\ldots,P_n$$

is implemented as follows:

```
"Link and load the new codes for $P_1,\ldots,P_n$";
for each P in $P_1,\ldots,P_n$ do
    LockProc (P)
end;
"Change the address entries of $P_1,\ldots,P_n$";
for each P in $P_1,\ldots,P_n$ do
    UnlockProc (P)
end;
```

The LockProc operation (defined in Figure 6-10) locks the procedure's address entry in PAT so that the procedure can not be called during the modification. Although procedures are not locked as an indivisible operation, the effects are equivalent for our purpose. That is, if the new version of an updated procedure is executed, it can never call the old version of another updated procedure. Furthermore, if the old version of an updated procedure is called and executed during the locking step (because they are not locked as an indivisible operation), the same old version can still be called and executed even if all the procedures are locked indivisibly depending on when they are locked.

```
LockProc (P):
   register adr;
   adr := "address of procedure P's entry in PAT";
   Lock (adr);

UnlockProc (P):
   register adr;
   adr := "address of procedure P's entry in PAT";
   Unlock (adr);
```

Figure 6-10. The LockProc and UnlockProc procedures.

```
var updateOK, timeout : signal;
   "Link and load the new codes for $P_1,...,P_n$"
   (* Computer total use count and set awaited bits *)
   TC := 1;
   for each Q in $Q_1,...,Q_m$ do
      LockProc (Q); "Set the awaited bit of procedure Q"
      Lock (TC); "Get the use count of Q and add to TC"
      Unlock (TC); UnlockProc (Q);
   end;
   Lock (TC); dec (TC);
   if TC = 0 then
      Unlock (TC); update $P_1,...,P_n$;
   else
      Unlock (TC);
      select
         when wait (updateOK) do update $P_1,...,P_n$; end;
      or
         when wait (timeout) do Print "update failed"; end;
      end;
   end;
   (* Clear awaited bits *)
   for each Q in $Q_1,...,Q_m$ do
      LockProc (Q);
      "Clear the awaited bit of procedure Q"
      UnlockProc (Q);
   end;
```

Figure 6-11. Update $P_1,...,P_n$ when $Q_1,...,Q_m$ idle.

Figure 6-11 outlines an implementation of the update command with a when-part. The total use count is computed from the use counts of the procedures specified in the when-part. If the total use count is zero, the procedures are updated immediately. The updated procedures can not be called while their address entries are changed as described in the previous case. Furthermore, the procedures specified in the when-part can not be called since their awaited bits are set and the total use count equals zero (see the call instructions in Figures 6-4 and 6-5). The procedures specified in the when-part can be called and executed when their awaited bits are cleared.

If the total use count is not zero, the dynamic modification process waits for the updateOK or timeout signal. While it is waiting, the procedures specified in the when-part can be called and executed as long as any one of the procedures is currently in use; that is, until the total use count becomes zero. We note that once the total use count is computed, the TC register always corresponds to the current total use count since it is updated from the procedure call and return instructions. The updateOK signal is sent by the user process that decreases the total use count to zero (see the return instruction in Figure 6-5). Whether the total use count has become zero is checked from the return instruction only (that is, it is checked only when it may become zero) to minimize the overhead and to be able to detect the condition without any delay (compared with, for example, some periodic checking scheme). Once the total use count becomes zero, the procedures whose awaited bits are set can not be executed until their awaited bits are cleared. The TC register is initialized to one so that the procedures specified in the when-part can be executed while the total use count is computed.

The timeout signal is included to prevent the dynamic modification process from waiting indefinitely. If updateOK is not signaled within some fixed time limit, the dynamic modification process is resumed and generates an "update failed" message to the programmer. Since the use counts of the procedures specified in the when-part are always correct and the awaited bits of the procedures are cleared, a new when-condition can be tried.

6.4.4. Updating Modules

Since a module can be replaced only when it is not in use, the module's exported procedures should be specified in the when-part of an update command when the module is updated. Furthermore, procedures that use changed exported types should also be specified in the when-part as discussed in Section 4.4.2. If the programmer omits these procedures, they are included in the when-part by the command interpreter.

Figure 6-12 outlines how a module is replaced, where the new version of the module contains a local data covert routine and exported type covert routines. When an update command is requested, the dynamic modification process checks that all the previous exported type conversions for variables of the module's exported type have been completed; if not, the module is not replaced. Otherwise, the new object code and data are linked and loaded into memory. As before, the awaited bits of the procedures specified in the when-part are set while their total use count is computed. Then the dynamic modification process waits for the total

```
"Link and load the new object code and data of M"
"Compute the total use count and set the awaited
 bits of Q1,...,Qn"
select
   when wait(updateOK) do
      "Lock the module's PAT and data frame address entries in MAT"
      "Change the module's PAT and data frame address entries if necessary"
      "Change the address entries in PAT"
      "Reallocate variables of exported types"
      "Execute the before-part of the local data convert routine"
      "Execute exported type convert routines for the variables in before-list's"
      "Set the lock bits of the variables in after-list's"
      "Clear the awaited bits of Q1,...,Qn"
      "Execute the after-part of the local data convert routine"
      "Unlock the module's PAT and data space address entries in MAT"
      "Create a process to execute exported type convert routines
       for the variables in after-list's"
   end;
or
   when wait (timeout) do
      "Clear the awaited bits of Q1,...,Qn"
      "Print "update failed""
   end;
end;
```

Figure 6-12. Update module M when $Q_1,...,Q_n$ idle.

use count to become zero or for the timeout signal.

If the when-condition is satisfied, the module's PAT and data frame address entries in MAT are locked to prevent other processes from using the module (see the external mode in Section 6.3.3). The PAT and data space are relocated if necessary. The procedure address entries of PAT are change to point to the new code segments. If the new version of the module contains a local data convert routine, its before-part is executed to initialize the new data structures.

If an exported type is changed, the indirect address entries of the variables (declared in other modules) of the exported type are changed to point to new space. If a type convert routine is provided, the type convert routine is executed for each variable in the before-list. If the type convert routine contains the after-list, the lock bits of the variables specified in the after-list are set to ensure that they are not referenced until properly initialized by an independent process created from the dynamic modification process. That process also clears their lock bits.

After the before-part's have been executed, the awaited bits of the procedures in the when-part are cleared so that they can be called and executed. If the local data convert routine contains the after-part, the after-part is now executed. Then, the module's PAT and data frame address entries in MAT are unlocked since the module has been successfully updated.

If the when-condition is not satisfied within some fixed time limit, the awaited bits of the procedures in the when-part are cleared and the appropriate error messages are printed. Furthermore, the portions of memory used by the new object code and data are reclaimed.

In summary, the following restrictions are observed by our implementation:

(1) A module can not be used while it is being updated since its data and PAT address entries are locked.

(2) The before-part of the local data convert routine and exported type convert routines for variables in before-list's are executed with the procedures in the when-part blocked.

(3) The after-part of the local data convert routine and exported type convert routines for variables in after-list's are executed without the procedures in the when-part blocked.

6.5. The Garbage Collection Process

In our system, code segments, PAT's, and data frames can be relocated at run-time without halting program execution and without creating dangling pointers. There are two reasons why they might have to be relocated. One reason is to move them into larger blocks of memory. For example, if new procedures are added to a module and if the module's PAT is not big enough, the PAT should be relocated. Another reason is to compact used portions of memory to provide a larger block of free memory.

To relocate a procedure code segment, the code segment is first copied into a new segment. Then, the procedure address entry in PAT is locked, changed to point to the new code segment, and unlocked. The old code segment can be reclaimed when its use count becomes zero since it can no longer be referenced.

The PAT of module m can be relocated to a block starting at an address, adr, as follows:

```
"Copy PAT into the block at adr";
m_entry := "the address of the module's old PAT";
Lock (m_entry);
MAT[m_entry] := adr;
Unlock (m_entry);
ChangeCB (m,adr);
"Reclaim the space used by the old PAT"
```

The current PAT of the module is first copied into a new block and then the module's

PAT address entry in MAT is changed to point to the new PAT. Finally, the CB register is changed to point to the new PAT if the current module is the module m when the ChangeCB instruction is executed (see ChangeCB in Section 6.3.4). Note that the ChangeCB instruction has to be executed after the entry in MAT has been modified.

Unlike the above two cases, a data frame should not be used while it is copied into a new frame for relocation. This restriction is necessary since the new data frame might become obsolete as soon as it is copied. Therefore, the data frame entry in MAT is first locked and then copied into the new frame. The data frame entry in MAT is then changed to point to the new frame before the entry is unlocked.

Since only logical addresses (except for return addresses) are stored in a stack, it can also be relocated. For example, the logical address of an actual argument to a "var" parameter is pushed on a process' stack. To prevent a stack from being modified while it is copied into a new location, the relocation of a process' stack should be done when it can not be used; that is, when the process is idle. If a stack is relocated, the saved values of the LP and HP registers are updated to point to the new stack. Furthermore, the saved values of the FP and SP registers are adjusted by a difference between the beginning addresses of the old and new stacks.

6.6. Summary

We have described how to implement our system. In particular, we have described a PST that represents hierarchical relations among the procedures and modules of a program and that contains the necessary information for each procedure and module. The compiler uses PST to recreate a compile-time environment to enforce type-checking. The command interpreter uses PST to check the validity of arguments to a user command, to retrieve source code for the editor and compiler, and to ensure the consistency of a program after an update command.

We have also described an architecture (that is, run-time program representation, addressing modes, and procedure call and return instructions) that supports the addition, deletion, and replacement of procedures and modules with the necessary synchronization and mutual exclusion. The architecture allows the relocation of code segments and data frames at any time. It also permits the detection of a when-condition as soon as the condition is satisfied. We note that the instructions and addressing modes described in this chapter can easily be emulated using conventional machine instructions with reasonable space and time overhead. We believe that hardware support through microprogramming can reduce execution time overhead comparable to conventional instructions and addressing modes.

CHAPTER 7

CONCLUSIONS

7.1. Summary

This dissertation has described a programming system that supports a single programmer modifying a running StarMod program. To modify procedures and modules dynamically, the programmer modifies and recompiles the source code of the procedures and modules and then requests the system to change the current code image to incorporate new code and data. Our approach has several advantages over traditional machine code patching. The current source code represents the machine code in execution. In particular, only changes that will leave the program consistent are allowed. Furthermore, the programmer need not know the details of the machine code generated by the compiler. Since the programmer only works with the source code, it is easier to determine what part of the program to modify and how such modification can be carried out. Finally, the core image of the running program is changed by the dynamic modification process so that changes are not susceptible to human errors.

Since future changes to a program might not be anticipated when the program is initially developed, our system supports any changes to a running program. That is, any procedures or modules can be added, deleted, and replaced. Also, any information stored in old data structures can be converted into new data structures. A running program may function unexpectedly if it is changed at an arbitrary moment. So our system allows the programmer to specify with an update command when the running program can be changed.

Since changes to a running program may require many procedures and modules be modified and updated, we have developed a methodology that can be used to carry out the changes in incremental steps. The methodology assists the programmer to find optimal update sequences based on update precedence relations among procedures and modules. The methodology and the advantages of updating fewer procedures and modules at a time are discussed in Chapter 5.

7.2. The Prototype System

In Chapter 6, we have described the program structure tree that represents hierarchical relations among procedures and modules. Each node of the program structure tree corresponds to a procedure or module and it contains source and object code, export and import lists, and a symbol table. The program structure tree is used to check the validity of arguments to user commands, to retrieve the source code of a requested procedure or module, and to reconstruct a compile-time environment. The command interpreter uses the export and import lists to verify that a program is consistent after an update command.

To support an implementation of our system, we have proposed an architecture; that is, the run-time program representation, addressing modes, and procedure call and return instructions. The architecture provides the necessary synchronization and mutual exclusion capability for the replacement of procedures and modules, the detection and sustenance of a when-condition, the conversion of data structures, and the relocation of code and data segments. The program is represented using the module and procedure address tables so that procedures and modules can efficiently added, deleted, and replaced. The execution time and space overhead of the architecture have been justified and seems reasonable for the supported capability.

A prototype dynamic modification system has been constructed to test the viability of our technique. It is running under the UNIX operating system on a Digital Equipment VAX. The compiler generates code that is essentially identical to that described in Chapter 6 and the code is interpreted. To allow concurrency between execution and modification, one process interprets code and the other dynamic modification process waits for an update command. When an update command is issued, the command interpreter wakes up the dynamic modification process, which then incorporates new code into the current code in memory without stopping the interpreter process. In our present implementation, procedures can be added, deleted, and replaced from a running program.

Our current implementation requires two separate terminals: one for the input and output of a running program and another for the programmer's interaction with the command interpreter. However, a multiple window environment like BRUWIN [37] would be ideal.

7.3. Future Research

In this section, we suggest problems for future research that are natural extensions of the system described in this dissertation.

7.3.1. Other Language Constructs

Our work has focused on changes to procedures and modules. To support all possible changes to a program, the system should be extended to allow dynamic changes to other language constructs, such as files, exception handlers, generic procedures and modules, and machine dependent code.

7.3.1.1. Files

As for the data representation of a module, files may need to be converted. Suppose our language is extended so that file F of type t is declared as follows:

F : file of t

With this file declaration, we assume that the following file access procedures are implicitly defined:

(1) **procedure** F.read (**var** x : t) - returns the next item of file F.

(2) **procedure** F.write (x : t) - stores x into file F.

(3) **procedure** F.eof : boolean - returns true if the last item has been read; false, otherwise.

(4) **procedure** F.reset - opens file F and initializes its current position to the beginning.

Suppose module M declares file f of type oldType. If module M is modified and file f is changed to contain items of type newType, file f can be restructured through the local data conversion routine of the new version of module M. For example, the new version of module M may contain the following local data conversion routine:

```
module M;
  ...
  f : file of newType;
  ...
  convert
    procedure ConvertType (x : M.oldType) : newType;
      begin
        (* convert a value of oldType to newType *)
      end ConvertType;
    var x : M.oldType; y : newType;
    before
      M.f.reset; f.reset;
      while not M.f.eof do
        M.f.read (x);
        y := ConvertType (x);
        f.write (y);
      end;
  end;
```

The local data conversion routine is executed when the module is not executing. If file f is very large, the module will be unavailable for a long time period. Furthermore, much time can be wasted to convert part of the file that may never be referenced again.

A different scheme for file conversion is to allow the programmer to define file handling procedures, read and write, with a new file declaration. The programmer defined read and write procedures provide the necessary type conversion. Therefore, files need not be really converted. For example, with the above new declaration of file f, the programmer defines the following read and write operations:

```
procedure f.read (var x : newType);
var y : oldType;
begin
  M.f.read (y);
  x := "coerce y to newType";
end f.read;

procedure f.write (x : newType);
var y : oldType;
begin
  y := "coerce x to oldType";
  M.f.write (y);
end f.write;
```

Items are always stored as type oldType; in particular, the items of type newType are converted into type oldType and then stored. For the read operation, an item of type oldType read from file f is converted into type newType. The main advantage of this method is that files need not be physically restructured. However, possible changes to files are limited since the old type needs to be converted into the new type and vice versa. The method can be extended to support arbitrary changes by storing the type information and data of each item. Here the read operation extracts the type information and then data using the type information. The data is converted into a requested type and returned. Unlike the previous scheme, only the old type needs to be converted into the new type but not vice versa.

If type information is to be stored with each item, space overhead can be reduced by storing type information with the first item of the new type. (Here type information and data should be distinguishable.) The space overhead can also be reduced by using a table of type information and pointers to items within a file. We need to study how files are used and modified to determine whether the above methods are sufficient for file conversion and which of them should be supported by the system.

7.3.1.2. Exception Handlers

To improve the reliability of programs, many programming languages, such as Ada [47], CLU [35], Mesa [38], allow exceptions to be raised and handled. A simple approach for the dynamic modification of exception handlers is to allow changes to an exception handler only when a procedure or module that contains the exception handler is modified. However, it may be necessary to be able to dynamically change only exception handlers. A further study of how exception conditions are raised and handled is needed to determine whether such generality is required.

7.3.1.3. Generic Procedures and Modules

Instead of requiring procedures or modules with similar properties to be declared separately, languages such as CLU [34], Alphard [53], EL 1 [49], and Ada [47] offer features for generic declaration of procedures or/and modules. Generic procedures and modules are parameterized procedure and module definitions. Instances of such procedures and modules can be created at compile time by providing parameters that are types or constants. The primary purpose of the generic procedures and modules are to reduce the size of the program text by factoring out dependencies on particular data types or constants. Furthermore, they also improve readability.

To extend the dynamic modification system to support changes to generic procedures and modules, the following questions need to be answered:

(1) Should the system allow changes be applied to some but not all instances of a changed generic procedure or module?

(2) Should all instances that need to be changed be updated at the same time or should the programmer specify when each instance can be updated?

We need to study on how generic procedures and modules are used to answer these questions and then extend the system to support the necessary features.

7.3.1.4. Machine Dependent Code

To provide complete but controlled access to the underlying machine, languages like Modula [50], Mesa [33], and Ada [47] permit code insertions and address specifications. These features are usually used to implement low-level programs, such as trap handlers, interrupt routines, device drivers. The system should be extended to allow the dynamic modification of machine dependent code.

7.3.2. The Program Structure

As Dijkstra has pointed out, the scope of present-day computations is far beyond the grasp of our unaided imagination. So "we must organize the computations in such a way that our limited powers are sufficient to guarantee that the computation will establish the desired effect" [13]. To support the dynamic modification of a program, the structure of the program should help the programmer to figure out not only how to change but also when to update the program. Therefore, the program should be designed and organized for change. In particular, the program should be structured to satisfy the following properties:

(1) The ripple effect of changes to procedures and modules should be limited. This can be achieved by following Parnas's information hiding policy [41] and restricting module and procedure dependency relations to form a tree [46].

(2) Procedures that are likely to be specified in a when-part should not be executed frequently. That is, a call graph of the program should have a high height with such procedures placed near leaf nodes.

We need to identify other properties that make programs easier to change dynamically. Then, we need to develop a methodology that will help the programmer to build programs that satisfy the desired properties.

7.3.3. The Programming Language

Our study of dynamic modification was based on the StarMod language. In particular, the language has been extended in a somewhat ad hoc manner to incorporate the dynamic modification concept explored in this dissertation. Instead of modifying an existing language definition, it seems better to design a programming language based on what we have found in this dissertation. For example, a new language might be designed to provide a uniform type conversion feature for the procedure's parameters and local variables, module's variables, files, and exported type variables.

7.3.4. Backup Mechanism

When a running program is modified, the programmer needs to be certain that the program will function as expected after the changes are made. To ensure the correctness of the proposed changes, they can first be tested using an experimental system that is a copy of the running program. Then, they can be dynamically applied to the running program. However, this testing may not be sufficient to guarantee the correctness of the new procedures and modules. In particular, it may not be possible to duplicate the operational environment of the running program.

Another way to ensure that the modified program will function as expected is to verify the correctness of the modified program. The verification technique is applicable regardless of whether the operational environment of the running program can be duplicated. However, informally verified procedures and modules might still contain errors as pointed out in [21,14]. Furthermore, although the development of formal verification systems has progressed rapidly in recent years, it is still not yet widely available.

To achieve the desired reliability of a running program, we may need to provide a means to roll back to the old version if the new version does not function as expected. The recovery block scheme for fault tolerant programs [27,42,2] can be embedded into our system. For example, when the new version of a procedure is created, we include an acceptance test. The acceptance test is a Boolean expression that is executed on exit from the new version. If it evaluates to false, all non-local variables altered by the new version are restored and then the old version is executed.

The problem of including the recovery block scheme into our system becomes complicated since the detection of an unacceptable result may require the roll back of previously accepted new versions. Furthermore, other user processes may have used the results of such previously accepted new versions; and therefore, such processes also need to be rolled back. Here, the conversational scheme that is to define a recovery structure common to the set of interacting processes can be used [42]. Although a language definition (that is, syntax and semantics) for the conversational scheme has been proposed as an extension to Concurrent Pascal [29], a further study is needed to show how the conversational scheme can be embedded into our approach to the dynamic modification of program. In particular, the usability and the implementation practicality of the conversation scheme require thorough consideration.

7.3.5. User Experience

We have shown that it is feasible to build a dynamic modification system. However, the feasibility does not suffice to justify the practicality of our system.

User experience is the most important factor in accessing the practicality of our system. To gain user experience, we need to build a "real" system so that application programs, such as operating systems, can be dynamically modified. Through such experience, we will be able to understand how hard it is to figure out proper when-conditions and whether more general when-conditions are necessary. Furthermore, we may find that the command interpreter needs to support more commands to help the programmer to modify a running program; for example, the human-readable display of possible effects of changes. Finally, we will be able to measure whether the space and execution time overhead of the proposed architecture is acceptable.

Although our system was designed to allow changes to a continuously running program, the system can be used to develop programs. For example, a program can first be constructed using minimal features and then incrementally changed to include more features. That is, the program can be implemented in a top-down manner dynamically. Another possible application of our system is for debugging programs. For example, a lot of time is spent in the cycle of debugging, correcting, recompiling, relinking, and restarting while developing a program. Sometimes it may take a long time to relink and reinitialize a program to test new procedures and modules. Therefore, a significant fraction of debugging time could be saved if the suspected procedures and modules can be replaced dynamically and then tested. Stoneman refers to the practice of patching machine code and retesting to reduce the debugging time for real-time programs [48].

To support program development using our system, it would be desirable to augment the system with a language based editor, such as the Cornell program synthesizer [45], POE [19], and IPE [36], and with an automatic incremental program verifier, such as Programmer's assistant [39], to do editing, program consistency checking, program verification, and code generation in parallel.

Appendix A

DYMOS User Command Syntax Summary

```
<command> ::= <edit> |
              <compile> |
              <delete> |
              <update>

   <edit> ::= edit <arg> [ from (current | provisional) ]

 <delete> ::= delete <arg list>

<compile> ::= compile <arg> [ (after | before) <arg> |
                          for (code | definition) ]

 <update> ::= update <arg list> [ delete <arg list> ]
                          [ when <arg list> idle [ within <limit> ] ]

<arg list> ::= <arg> {, <arg> }

    <arg> ::= <id> {. <id>}

     <id> ::= procedure name |
              module name

  <limit> ::= real number
```

Appendix B

The Simple On-Line Banking System

```
(*
  This program is the complete listing of the on-line banking system described
  in Chapter 1.
*)
module main (* BankingSystem *);
  const minAccountNo = 100; maxAccountNo = 200;
      maxStringSize = 80; endOfString = 0s;
  type string = array 1 : maxStringSize of char;

  (*
    The BookKeeper module provides routines to manage available account
    numbers and to fetch and to change information associated with each account.
  *)
  module BookKeeper;
    export StoreName, GetName, GetNewAccount, AdjustBalance, GetBalance;
    import minAccountNo, maxAccountNo, string, endOfString;

    (*
      The NameStorage module to store customer names, where each name is
      stored in an array of 40 characters.
    *)
    module NameStorage;
      export ChangeIntoName, ChangeIntoString, nameType;
      import string, endOfString;
      const nameSize = 40;
      type  nameType = array 1 : nameSize of char;

      (* This procedure converts a string into a name. *)
      procedure ChangeIntoName (var name : nameType; str : string);
      var i : integer;
      begin
        i := 1;
        loop
          name[i] := str[i];
          when str[i] = endOfString do exit;
          inc (i);
        end;
      end ChangeIntoName;

      (* This procedure converts a name into a string. *)
      procedure ChangeIntoString (name : nameType; var str: string);
      var i : integer;
      begin
        i := 1;
        loop
          str[i] := name[i];
          when name[i] = endOfString do exit;
      inc (i);
    end;
  end ChangeIntoString;
end NameStorage;

var data : array minAccountNo : maxAccountNo of
             record
               name : nameType;
               balance : integer;
             end;
   availAccountNo : integer;

(* This procedure stores a name into a given account. *)
procedure StoreName (acnt : integer; str : string);
begin
  ChangeIntoName (data[acnt].name, str);
end StoreName;

(* This procedure returns the name of a given account. *)
procedure GetName (acnt : integer; var str: string);
begin
  ChangeIntoString (data[acnt].name, str);
end GetName;

(* This procedure returns the next available account number. *)
procedure GetNewAccount : integer;
begin
   GetNewAccount := availAccountNo;
   inc (availAccountNo);
end GetNewAccount;

(* This procedure adjust the balance of an account by a given amount. *)
procedure AdjustBalance (acnt, amt : integer);
begin
   inc (data[acnt].balance, amt);
end AdjustBalance;

(* This procedure returns the balance of a given account. *)
procedure GetBalance (acnt : integer) : integer;
begin
   GetBalance := data[acnt].balance;
end GetBalance;

begin
   (* Initialize the available account number. *)
   availAccountNo := minAccountNo;
end BookKeeper;

(*
  The RequestHandler module provides routines to handle customer's requests.
  The format of a request is described in Chapter 1.
*)
module RequestHandler;
```

```
export ProcessRequest;
import string, endOfString, GetName, StoreName,
    GetNewAccount, AdjustBalance, GetBalance;
type  transType = (Deposit, Withdraw, Open, Print);

(*
  The InputOutput module provides routines to read requests
  and to write requested information to the user's terminal.
*)
module InputOutput;
  export WriteLine, ReadTransType, ReadName, PrintName,
      ReadAccountNo, PrintAccountNo, ReadAmount, PrintBalance;
  import string, endOfString, transType, Deposit, Withdraw, Open, Print;

  const  (* System constants for i/o routines *)
      writeln = 2; write = 3; sysCall = 6;
      read = 18; standardInput = 0;
      newLine = 12s; bufferSize = 79;

  var buffer : array 0 : bufferSize of char;
     linePosition : integer;

  (* This procedure outputs a new line onto the terminal. *)
  procedure WriteLine;
  begin sys (writeln,' ') end WriteLine;

  (* This procedure reads in one line from the terminal into 'buffer'. *)
  procedure ReadLine;
  var lineLength,i : integer;
  begin
    lineLength := sys (sysCall, read, standardInput, buffer, bufferSize);
    linePosition := 0;
    while buffer[linePosition] = ' ' do inc (linePosition) end;
  end ReadLine;

  (* This procedure returns the type of the current request. *)
  procedure ReadTransType (var trans: transType);
  var i : integer;
  begin
    sys (write,'Action: '); ReadLine; i := linePosition;
    if (buffer[i] = 'd') and (buffer[i+1] = 'e') and (buffer[i+2] = 'p') and
      (buffer[i+3] = 'o') and (buffer[i+4] = 's') and (buffer[i+5] = 'i') and
      (buffer[i+6] = 't') then
          trans := Deposit
    elsif (buffer[i] = 'w') and (buffer[i+1] = 'i') and (buffer[i+2] = 't') and
      (buffer[i+3] = 'h') and (buffer[i+4] = 'd') and (buffer[i+5] = 'r') and
      (buffer[i+6] = 'a') and (buffer[i+7] = 'w') then
          trans := Withdraw
    elsif (buffer[i] = 'o') and (buffer[i+1] = 'p') and
      (buffer[i+2] = 'e') and (buffer[i+3] = 'n') then
          trans := Open;
    elsif (buffer[i] = 'p') and (buffer[i+1] = 'r') and
      (buffer[i+2] = 'i') and (buffer[i+3] = 'n') and
    (buffer[i+4] = 't') then
          trans := Print;
  else
          sys (writeln,'*** Unknown command ***');
          ReadTransType (trans);
  end;
end ReadTransType;

(* This procedure returns a customer name included in 'buffer'. *)
procedure ReadName (var name : string);
var i : integer;
begin
  sys (write,'Name: '); ReadLine; i := 1;
  while buffer[linePosition] <> newLine do
    name[i] := buffer[linePosition];
    inc (i); inc (linePosition);
  end;
  name[i] := endOfString;
end ReadName;

(* This procedure checks whether a given character is a digit. *)
procedure IsDigit (ch : char) : boolean;
begin
  if ('0' <= buffer[linePosition]) and (buffer[linePosition] <= '9') then
     IsDigit := true;
  else
     IsDigit := false;
  end;
end IsDigit;

(* This procedure returns a number stored in 'buffer'. *)
procedure ReadNumber (var val : integer);
const ordOf0 = ord('0');
var n : integer; minusFlag : boolean;
begin
  ReadLine;
  if buffer[linePosition] = '-' then
     minusFlag := true; inc (linePosition);
  else
     minusFlag := false;
  end;
  val := 0;
  if not IsDigit (buffer[linePosition]) then
     sys (write,'Please enter a number: '); ReadNumber (val);
  else
     while IsDigit (buffer[linePosition]) do
        n := ord(buffer[linePosition]) - ordOf0;
        val := val * 10 + n; inc (linePosition);
     end;
     if minusFlag then val := - val end;
  end;
end ReadNumber;
```

```
 (* This procedure prints a name on the terminal. *)
 procedure PrintName (name : string);
 begin sys (write, 'Name %s, ',name) end PrintName;

 (* This procedure asks and reads an account number from the terminal. *)
 procedure ReadAccountNo (var acnt : integer);
 begin
   sys (write,'Account No.: '); ReadNumber (acnt);
 end ReadAccountNo;

 (* This procedure writes an account number on the terminal. *)
 procedure PrintAccountNo (acnt : integer);
 begin sys (write, 'Account No. %d, ', acnt) end PrintAccountNo;

 (* This procedure asks and reads an amount from the terminal. *)
 procedure ReadAmount (var amt : integer );
 begin sys (write,'Amount: '); ReadNumber (amt) end ReadAmount;

 (* This procedure prints the balance on the terminal. *)
 procedure PrintBalance (amt : integer );
 begin sys (write, 'Balance %d ', amt) end PrintBalance;

end InputOutput;

(* This procedure handles the Print request. *)
procedure PrintAccount;
var name : string; acnt, amt : integer;
begin
  ReadAccountNo (acnt); PrintAccountNo (acnt);
  GetName (acnt, name); PrintName (name);
  amt := GetBalance (acnt); PrintBalance (amt); WriteLine;
end PrintAccount;

(* This procedure handles the Open request. *)
procedure OpenAccount;
var acnt : integer; name : string;
begin
  acnt := GetNewAccount; ReadName (name); StoreName (acnt, name);
  PrintAccountNo (acnt); WriteLine;
end OpenAccount;

(*
  This procedure continuously reads one request at a time
  and takes an appropriate action.
*)
procedure ProcessRequest;
var trans: transType; acnt, amt : integer;
begin
  loop
    ReadTransType (trans);
    case trans of
      Deposit, Withdraw:
        begin ReadAccountNo (acnt); ReadAmount (amt);
```

```
                AdjustBalance (acnt, amt)
            end;
          Open:  begin OpenAccount; end;
          Print: begin PrintAccount; end;
          otherwise: begin (* Print error messages *) end;
        end; (* case *)
      end; (* loop *)
    end ProcessRequest;

  end RequestHandler;

begin
   ProcessRequest;
end main. (* BankingSystem *)
```

Appendix C

The Simple On-Line Banking System with Changes Described in Chapter 4

```
module main (* BankingSystem *);

  (* This part is not changed. *)

  (* The BookKeeper and NameStorage modules as modified in Figure 4-8. *)
  module BookKeeper;
    export StoreName, GetName, GetNewAccount, AdjustBalance, GetBalance;
    import minAccountNo, maxAccountNo, string, endOfString;

    module NameStorage;
      export ChangeIntoName, ChangeIntoString, nameType;
      import string, endOfString;
      const maxNamePoolSize = 80;
      type  nameType = record
                         start, length : integer;
                       end;
      var  namePool : array 1 : maxNamePoolSize of char;
           availPtrNamePool : integer;

      procedure ChangeIntoName (var name : nameType; str : string);
      var i : integer;
      begin
        i := 1; name.start := availPtrNamePool;
        loop
          namePool[availPtrNamePool] := str[i];
          when str[i] = endOfString do exit;
          inc (i); inc (availPtrNamePool);
        end;
        name.length := i - 1;
      end ChangeIntoName;

      procedure ChangeIntoString (name : nameType; var str: string);
      var i, j : integer;
      begin
        i := 1; j := name.start;
        loop
          str[i] := namePool[j];
          when i = name.length do exit;
          inc (i); inc (j);
        end;
        str[i+1] := endOfString;
      end ChangeIntoString;
    begin
      availPtrNamePool := 1;
    end NameStorage;

    var data : array minAccountNo : maxAccountNo of
```

```
              record
                name : nameType;
                balance : integer;
              end;
     availAccountNo : integer;

  (* Procedures StoreName, GetName, GetNewAccount have not been changed. *)

  (* Procedure AdjustBalance as modified in Figure 1-4. *)
  procedure AdjustBalance (acnt, amt : integer);
  const  writeln = 2; maxInteger = 32767;
  begin
    if (amt < 0) and (data[acnt].balance < -amt) then
       sys (writeln,'Overdraw not allowed');
    elsif (amt > 0) and (maxInteger-amt < data[acnt].balance) then
       sys (writeln,'Exceeding account limit not allowed');
    else
       inc (data[acnt].balance, amt);
    end;
  end AdjustBalance;

  (* Procedure GetBalance as modified in Figure 4-1. *)
  procedure GetBalance (acnt : integer; var name: string; var amt: integer);
  begin
     GetName (acnt, name); amt := data[acnt].balance;
  end GetBalance;

begin
   (* This part has not been changed. *)
end BookKeeper;

module RequestHandler;

   (* This part has not been changed. *)

   module InputOutput;
     (* Module InputOutput has not been changed. *)
   end InputOutput;

   (* Procedure PrintAccount as modified in Figure 4-1. *)
   procedure PrintAccount;
   var name : string; acnt, amt : integer;
   begin
     ReadAccountNo (acnt); PrintAccountNo (acnt); GetBalance (acnt, name, amt);
     PrintName (name); PrintBalance (amt); WriteLine;
   end PrintAccount;

   (* Procedure OpenAccount has not been changed. *)

   (* The new Processtrans procedure as added in Figure 4-2. *)
   procedure ProcessTrans (trans: transType; acnt, amt: integer);
   const writeln = 2;
   begin
```

```
      if amt < 0 then sys (writeln,'Use positive number');
      elsif trans = Deposit then AdjustBalance (acnt, amt);
      else AdjustBalance (acnt, -amt);
      end;
    end ProcessTrans;

    (* Procedure ProcessRequest as modified in Figure 4-4. *)
    procedure ProcessRequest;
    var trans: transType; acnt, amt : integer;
    begin
      loop
        ReadTransType (trans);
        case trans of
          Deposit, Withdraw:
            begin ReadAccountNo (acnt); ReadAmount (amt);
                ProcessTrans (trans, acnt, amt)
            end;
          Open:  begin OpenAccount; end;
          Print: begin PrintAccount; end;
          otherwise: begin (* Print error messages *) end;
        end; (* case *)
      end; (* loop *)
    end ProcessRequest;

  end RequestHandler;

begin
  (* This part has not been changed. *)
end main. (* BankingSystem *)
```

Bibliography

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley Publishing Company (1974).

[2] T. Anderson and R. Kerr, "Recovery Blocks in Action: A System Supporting High Reliability," *Proc. 2nd Int. Conf. on Soft. Eng.*, pp. 447-457 (1976).

[3] P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE-TSE SE-1*, 2, pp. 199-207 (June 1975).

[4] P. Cashin, M.L. Joliat, R.F. Kamel, and D.M. Lasker, "Experience with a Modula Typed Language: PROTEL," *Proc. 5th Int. Conf. on Soft. Eng.*, (March 1981).

[5] B.G. Claybrook, "A Specification Method for Specifying Data and Procedural Abstractions," *IEEE-TSE SE-8*, 5, pp. 449-459 (September 1982).

[6] R.P. Cook and S.J. Scalpone, *An Introduction to StarMod for Pascal Users*, UW-Madison Tech. Rep. 372 (1979).

[7] R.P. Cook and I. Lee, *An Extensible Stack-Oriented Architecture for a High-Level Language Machine*, UW-Madison Tech. Rep. 397 (August 1980).

[8] R.P. Cook, "*Mod--A Language for Distributed Programming," *IEEE-TSE SE-6*, 6, pp. 563-571 (November 1980).

[9] R.P. Cook and I. Lee, "A Contextual Analysis of Pascal Programs," *Software--Practice and Experience 12*, 2, pp. 195-203 (February 1982).

[10] R.P. Cook, R.H. Gerber, and T.J. LeBlanc, Kernel Design for Concurrent Programming, Submitted to Software--Practice and Experience (1982).

[11] R.P. Cook and T.J. LeBlanc, "A Symbol Table Abstraction to Implement Languages with Explicit Scope Control," *IEEE-TSE SE-9*, 1, pp. 8-12 (January 1983).

[12] O. Dahl and K. Nygaard, "SIMULA - An ALGOL Based Simulation Language," *Comm. ACM* 9, 9, pp. 671-678 (September 1966).

[13] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming*, Academic Press, London (1972).

[14] R.A. DeMillo, R.J. Lipton, and A.J. Perlis, "Social Processes and Proofs of Theorems and Programs," *Comm. ACM 22*, 5, pp. 271-280 (May 1979).

[15] F. DeRemer and H.H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE-TSE SE-2*, 2, pp. 80-86 (June 1976).

[16] G.W. Ernst and W.F. Ogden, "Specification of Abstract Data Types in Modula," *ACM-TOPLAS* 2, 4, pp. 522-543 (October 1980).

[17] R.S. Fabry, "Capability-based addressing," *Comm. ACM 17*, 7, pp. 403-412 (July 1974).

[18] R.S. Fabry, "How to design a system in which modules can be changed on the fly," *Proc. 2nd Int. Conf. on Soft. Eng.*, pp. 470-476 (1976).

[19] C.N. Fischer, G. Johnson, and J. Mauney, *An Introduction to Release 1 of Editor Allan Poe*, UW-Madison Tech. Rep. 451 (1981).

[20] D.G. Foxall, M.L. Joliat, R.F. Kamel, and J.J. Miceli, "PROTEL: A High Level Language for Telephony," *Proc. 3rd Int. Comp. Soft. and Appl. Conf.*, (November 1979).

[21] S.L. Gerhart and L. Yelowitz, "Observations of Fallibility in Applications of Medern Programming Methodologies," *IEEE-TSE SE-2*, 3, (September 1976).

[22] R.L. Glass, "Patching is Alive and, Lamentably, Thriving in the Real-Time World," *SIGPLAN Notices 13*, 3, pp. 25-28 (March 1978).

[23] H. Goullon, R. Isle, and K. Lohr, "Dynamic Restructuring in an Experimental Operating System," *IEEE-TSE SE-4*, 4 , pp. 298-307 (July 1978).

[24] M. Herlihy and B. Liskov, "Communicating Abstract Values in Messages," Computation Structures Group Memo 200, MIT Laboratory of Computer Science, Cambridge, Ma. (October 1980).

[25] C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica 1*, pp. 271-281 (1972).

[26] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," *Comm. ACM 17*, 10, pp. 549-556 (October 1974).

[27] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell, "A Program Structure for Error Detection and Recovery," in *Operating Systems, Lecture Notes in Computer Science No. 16*, Spring-Verlag (1974).

[28] R.K. Johnsson and J.D. Wick, "An Overview of the Mesa Processor Architecture," *Symposium on Architectural Support for Prog. Lang. and Oper. Sys.*, pp. 20-29 (March 1982).

[29] K.H. Kim, "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE-TSE SE-8*, 3, pp. 189-197 (May 1982).

[30] P.M. Kogge, "An Architectural Trail to Threaded-Code System," *Computer 15*, 3, pp. 22-32 (March 1982).

[31] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.L. Popek, "Report on the Programming Language Euclid," *SIGPLAN Notices 12*, 2, (February 1977).

[32] D.M. Lasker, "Module Structure in an Evolving Family of Real Time Systems," *Proc. 4th Int. Conf. on Soft. Eng.*, (September 1979).

[33] H.C. Lauer and E.H. Satterthwaite, "The Impact of Mesa on System Design," *Proc. 4th Int. Conf. on Soft. Eng.*, pp. 174-182 (September 1979).

[34] B.H. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *Comm. ACM 20*, 8, pp. 564-576 (August 1977).

[35] B.H. Liskov and A. Snyder, "Exception Handling in CLU," *IEEE-TSE SE-5*, 6, pp. 546-558 (November 1979).

[36] R. Medina-Mora and P.H. Feiler, "An Incremental Programming Environment," *IEEE-TSE SE-7*, 5, pp. 472-482 (September 1981).

[37] N. Meyrowitz and M. Moser, "BRUWIN: An Adaptable Design Strategy for Window Manager/Virtual Terminal System," *Proc. 8th Symp. on Oper. Sys. Principles*, pp. 180-189 (December 1981).

[38] J.G. Mitchell, W. Maybury, and R.E. Sweet, "Mesa Language Manual Version 5.0," CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, California (1979).

[39] M.S. Moriconi, "A Designer/Verifier's Assistant," *IEEE-TSE SE-5*, 4, pp. 387-401 (July 1979).

[40] D.L. Parnas, "On the criteria to be used in decomposing systems into modules," *Comm. ACM 15*, 12, pp. 1053-1058 (December 1972).

[41] D.L. Parnas, "A technique for software module specification with examples," *Comm. ACM 15*, 5, pp. 330-336 (May 1972).

[42] B. Randell, "System Structure for Software Fault Tolerance," *IEEE-TSE SE-1*, 2, pp. 220-232 (June 1975).

[43] A. Rudmik and B.G. Moore, "An Efficient Separate Compilation Strategy for Very Large Programs," *SIGPLAN Notices 17*, 6, pp. 301-307 (June 1982).

[44] A.S. Tanenbaum, "Implications of Structured Programming for Machine Architecture," *Comm. ACM 21*, 3, pp. 237-246 (March 1978).

[45] T. Teitelbaum, T. Reps, and S. Horwitz, "The why and wherefore of the Cornell Program Synthesizer," *SIGPLAN Notices 16*, 6, pp. 8-16 (1981).

[46] J. Turner, "The Structure of Modula Programs," *Comm. ACM 23*, 5, pp. 272-277 (May 1980).

[47] U.S. Department of Defense, Reference Manual for the Ada Programming Language. (July 1980).

[48] U.S. Department of Defense, Stoneman: Requirements for Ada Programming Support Environments. (February 1980).

[49] B. Wegbreit, "The Treatment of Data Types in EL 1," *Comm. ACM 17*, 5, pp. 251-264 (May 1974).

[50] N. Wirth, "Modula: A Language for Modular Multiprogramming," *Software Practice and Experience 7*, 1, pp. 3-35 (January 1977).

[51] N. Wirth, "Modula-2," Report 36, Institut fur Informatik, Zurich, ETH. (March 1980).

[52] N. Wirth, "Lilith: A Personal Computer for the Software Engineer," *Proc. 5th Int. Conf. on Soft. Eng.*, pp. 2-15 (1981).

[53] W.A. Wulf, R.L. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs," *IEEE-TSE SE-2*, 4, pp. 253-265 (December 1976).