1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - Votre référence

Our file - Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

A Meta-Model for Dynamic Change Management

by

Matthew R. Kasun, B.Eng.

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements of the degree of**

Master of Engineering

**Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
28 May 1993**

**© copyright
1993 Matthew R. Kasun**

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-89861-5

Canada

The undersigned hereby recommend to the Faculty of Graduate Studies and Research acceptance of the thesis

A Meta-Model for Dynamic Change Management

submitted by Matthew R. Kasun, B. Eng., in partial fulfillment of the requirements for the degree of Master of Engineering.

Chair

Department of Systems and
Computer Engineering

Thesis Supervisor

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
August 12, 1993

Abstract

A method of dynamically updating software is required for long running systems. Previous research in this area has produced solutions which are only applicable in specific environments. This thesis examines the theory of dynamic change management. The theory can be used to explain existing techniques, hypothesize new ones, and explore limitations.

The foundation of the theory is a model of an executing program. A program is modeled as a collection of extended state machines (modules) which communicate via message passing. Change is implemented by the execution of a series of change primitives. A meta-level module is required to control the execution of change primitives.

Dynamic change must be controlled to ensure that the program continues to execute correctly during the implementation of a modification. The control is exercised by a change management system which identifies the change primitives to be executed, and determines the order and timing of execution. The correct execution of a program is dependent upon the consistency of a program's code, data variables, and communication links. Consistency does not have to be maintained at all times, although this is the more elegant solution. The change management system can prevent an inconsistency from manifesting itself into an error by controlling the execution of the program.

The theory of dynamic change is validated by describing the solutions of previous researchers in terms of the model. The use of the model simplifies the comparison of different solutions and highlights their limitations.

Acknowledgments

I would like to first thank Mike Petras of Bell Northern Research for providing me with some insight to the real life problems that one faces when trying to dynamically update an executing program.

I would next like to thank Gerald Karam, my thesis advisor. His encouragement and numerous suggestions made it possible to turn a disjointed collection of random thoughts into something approaching readability.

I would also like to thank my family, especially my wife Janet, who patiently listened to my ramblings about communicating state machines, change primitives and dining philosophers; topics quite foreign to an accountant and future law student.

Finally I would like to thank the Department of National Defence for sponsoring this graduate work.

Table of Contents

Chapter 1 - Introduction	1
1.1 Motivation	1
1.2 Thesis Objectives	3
1.3 Thesis Summary	4
1.4 Thesis Overview	5
Chapter 2 - Previous Research	6
2.1 Changing Modules on the Fly	6
2.2 Dynamic Alterable System	7
2.3 DMERT	8
2.4 Dynamic Module Replacement	8
2.5 Dynamic Modification System	9
2.6 Object Oriented Languages	10
2.7 Ada	10
2.9 PROTEL	11
2.10 PODUS	11
2.11 Kramer & Magee	12
2.12 Summary	13
Chapter 3 - Model of a Computer Program	14
3.1 General	14
3.2 Modules	15
3.3 Intermodule Communications	17
3.3.1 Communication Primitives	17
3.3.2 Logical and Physical Communication Models	18
3.3.2.1 Interprocess Communications	18
3.3.2.2 Intraprocess Communications	21
3.3.2.3 Logical and Physical Representations of Communication Links	23
3.3.3 Multiple Instances of Modules	24
3.4 Operating System and Meta-Level Modules	24
3.5 Summary of the Program Model	25
Chapter 4 - The Implementation of Change	27
4.1 Modifications	27
4.2 Implementation of Modifications	28
4.2.1 Module Updates	29

4.2.1.1 Module Replacement.....	30
4.2.1.2 Module Overwriting.....	30
4.2.2 Addition and Deletion of Modules	31
4.2.3 Reconfiguration of Communication Links	33
4.2.4 Control State Updates	37
4.3 Implementation of Change - Summary.....	39
Chapter 5 - Management of Change.....	40
5.1 Introduction.....	40
5.2 Correctly Executing Program	40
5.2.1 Consistency	41
5.3 The Change Management Process	42
5.4 Management of Code Consistency.....	43
5.5 Consistency of Message Transmissions.....	45
5.5.1 Consistency of Communication Links	46
5.5.2 Module Interface Consistency.....	49
5.6 Data Consistency	52
5.6.1 Inconsistent Data State	54
5.6.2 Management of Data Consistency.....	55
5.6.2.1 Delaying Execution	55
5.6.2.2 Updating to a Consistent Value	56
5.6.2.3 Isolation.....	57
5.6.3 Managing Data Inconsistencies	58
5.7 The Change Management Process - Summary	59
Chapter 6 - Model Validation	60
6.1 Introduction.....	60
6.2 Module Replacement.....	61
6.2.1 Operating System Solutions.....	61
6.2.1.1 DMERT.....	61
6.2.1.2 Dynamic Module Replacement	62
6.2.1.3 Data Restructuring	63
6.2.1.3.1 Changing Modules on the Fly.....	63
6.2.1.3.2 Dynamic Alterable System	64
6.2.1.4 PODUS.....	66
6.2.2 Language Based Solutions.....	68
6.2.2.1 Object Oriented Languages	68
6.2.2.2 Secure on the Fly Modifications	69
6.2.2.3 Ada.....	69
6.3 Change Management Systems	69
6.3.1 Dynamic Modification System	69
6.3.2 Kramer & Magee.....	73
6.4 Summary.....	74

Chapter 7 - Conclusions	76
7.1 Theory of Dynamic Change.....	76
7.1.1 Model.....	76
7.1.2 Change Primitives.....	77
7.1.3 Consistency	78
7.2 Validation	78
7.3 Future Work	79
Appendix A - Updating the Main Procedure of a Program	84
Appendix B - Translation Modules	87

List of Figures

2.1 - Changing Modules on the Fly.....	7
2.2 - DAS Module	8
3.1 - Modules	15
3.2 - Module.....	16
3.3 - Abstract Memory Module.....	19
3.4 - Communications With Abstract Module.....	20
3.5 - Rendezvous	20
3.6 - Indirect Procedure Calls.....	22
3.7 - Indirect Procedure Calls - Pseudo Code	22
3.7 - Communication Links	23
4.1 - Updating the Originator	34
4.2 - Updating the Recipient.....	35
4.3 - Reconfiguration of Communication Links by Module Replacement	36
4.4 - Reconfiguration of Communication Links by Overwriting	37
5.1 - Modification	43
5.2 - Management of Code Inconsistencies.....	45
5.3 - Management of Communication Link Inconsistencies.....	48
5.4 - Translation Module.....	50
5.5 - Managing Interface Consistency.....	51
5.6 - Airline Reservation System	53
5.7 - Airline Reservation Code	54
5.8 - Management of Data Inconsistencies.....	58
5.9 - Main Functions of Change Management Process.....	59
6.1 - DMERT	62
6.2 - Changing Modules on the Fly.....	63
6.3 - Repcode Operation	65
6.4 - Chlink Operation	65
6.5 - Repdat Operation.....	66
6.6 - DYMOS Execution Control.....	71
6.7 - Module Interface Consistency	72
6.8 - Managing Data Consistency.....	72
6.9 - Configuration Management System.....	73
A.1 - Updating Return Address.....	85
A.2 - Replacing Main Procedure	86
B.1 Update to Sort Module.....	87

Chapter 1

Introduction

1.1 Motivation

The maintenance of software is a continuous process. Updates to computer programs are required for a variety of reasons including bug fixes, the addition of new capabilities, and the improvement of existing capabilities. Regardless of the reason(s) for the change(s), once a modification has been designed, coded, compiled and tested, the current version of the program must be replaced by the updated version. In a *static modification scheme* the execution of the current version is halted, the new version is loaded into memory, and the program is restarted. This approach is perfectly acceptable for programs which have a short run time as the update can be scheduled to coincide with a planned shutdown. However, this approach is not desirable for programs executing in long running systems (systems which run continuously for months or even years, e.g., telephone switching networks, airline reservation systems, automatic banking networks, satellite systems, and nuclear power plant systems). There may be economic reasons which preclude shutdown or the system may contain long term state information which must be preserved. Unless the state information is stored in non-volatile memory it will be lost. An alternative to shutting down the system is to use a *dynamic modification scheme*, i.e., update the program while it continues to execute.

As in most system issues, there are both hardware and software approaches to implementing dynamic change. A hardware approach requires a multi-processor system: (1) the new version of the program is loaded on a separate CPU and set of peripherals; (2)

control is transferred to the new version and the old version is shutdown; and (3) the two sets of hardware are then re-synchronized. In long running systems, the extra hardware is normally provided for fault tolerance reasons or to improve performance. During the update the extra hardware is not available for its intended purpose and thus overall system performance is degraded during the update. Also, the transfer of state information is often a complex problem. A software approach eliminates the need for extra hardware; if extra hardware is available it can be employed in its primary role - fault tolerance or performance enhancement. This dissertation examines the software approach to dynamic change management.

The dynamic modification of a computer program can be divided into two separate activities; the *implementation of change* and the *management of change*. A *computer program* is a set of machine readable instructions and data variables stored in memory (either volatile or non-volatile) locations. Thus, in order to implement a change, these memory locations must be overwritten with new values. This activity is relatively straightforward, as a separate process can be loaded which overwrites the locations.¹ The real challenge in a dynamic modification scheme is the second activity; ensuring that the program will execute correctly while these memory locations are being overwritten.

Previous research in this area has concentrated on solutions for dynamically modifying a program in a specific environment: either a particular operating system or a particular programming language or class of languages. In several cases, a new operating system and/or programming language was developed [12, 14, 24, 26] to support dynamic change management. By concentrating on a specific environment, a feature of the environment, such as indirect addressing [12, 18, 24, 26], data locking mechanisms [8, 12, 18] or deferred classes [25], can be exploited to simplify the implementation and management of change. However, a solution which exploits a particular feature of the operating system or programming language cannot be applied in an environment that does not support that feature.

In addition to being limited to a particular environment, the majority of previous solutions also restrict the types of changes which are supported. For example, only two solutions [18, 24] support changes to the parameters of a procedure call. Restricting the

¹ The implementation of change does not necessarily require a separate process; the implementation of change could be controlled by the operating system or by the program being updated.

types of changes supported simplifies the implementation and management of change, but limits the utility of the solution.

The third major shortcoming that exists in most of the previous solutions, is the requirement that the original version of the program (i.e., the version to be updated) be designed for dynamic change thus adding significant overhead to the normal execution of the program. The most dramatic example of this is the solution proposed by Kramer & Magee [15]. In their Evolving Philosophers example, the activities of a philosopher are governed by a set of seven rules. In order to accommodate dynamic change, one of these rules must be modified and a further six rules added.

1.2 Thesis Objectives

The objective of this thesis is to develop the theory of dynamic change, i.e., to develop a dynamic change model that is independent of a programming language and operating system. The model of dynamic change is a *meta-model* that consists of:

- (1) a generic representation of a computer program, the *model of a computer program*;
- (2) a description of the techniques for implementing various types of changes, the *model of the change implementation process*; and
- (3) an identification of the additional activities which must be performed and coordinated in order to ensure that program being updated continues to execute correctly during the update, a *model of the change management process*.

Dynamic change theory identifies options for implementing various types of changes to a program, thus, it allows a dynamic change management system to be engineered for a particular environment. Certain types of changes are easier to implement if the target environment contains a specific feature. In this case, the feature may be exploited to simplify the design of the change management system. If not, the designer of the change management system must decide whether to add the feature to the environment, to use a different technique for implementing that type of change, or to not support that type of change.

1.3 Thesis Summary

A computer program may consist of one or more processes² and is modeled as a collection of communicating modules where each module is an extended finite state machine. The model includes the program and its environment; i.e., the operating system and any hardware (hardware is encapsulated in *abstract modules*) with which the program directly reacts. Modules communicate via synchronized message passing. The logical flow of information between two program modules normally involves a physical flow of information among three modules: the two program modules and an intermediate communication handler module provided by the operating system. Thus, two representations of a program are possible; a *logical communications model* and a *physical communication model*. A modification of a computer program must be defined in terms the logical model but the implementation of change requires the physical model.

Some modules of the operating system perform functions in which program modules are treated as data; i.e., they are *meta-level modules*. The dynamic change management system also contains meta-level modules. Modules which simply interact with other modules but do not manipulate other modules are referred to as *object-level modules*.

A modification to a computer program can be broken down into a set of *modification operations* and each modification operation is implemented by executing a series of *change primitives*. The change primitives are: *overwriting a data variable*; *loading a module*, and *removing a module*. The change management system controls the execution of change primitives but does not necessarily execute them. It may delegate the execution to the operating system or to a program module.

The other role of the change management system is to ensure that the program continues to execute correctly during the implementation of the modification. An incorrectly executing program results from inconsistencies in: (1) the code executed by the program; (2) the state of the program; (3) the interface between modules; or (4) the configuration of communication links between modules.

² It can be argued that each process is a separate program, however, in this dissertation the term program refers to all process.

The execution of a change primitive can introduce an inconsistency, however the presence of an inconsistency does not necessarily result in the immediate incorrect execution of the program. In a properly designed modification, the inconsistency will be eliminated by the execution of subsequent change primitives and therefore, if the change management system can restrict the execution of certain components, the inconsistency can be prevented from manifesting itself. The other, more elegant solution is to eliminate the possibility of an inconsistency by performing additional operations and restructuring the sequence in which change primitives are executed.

The theory of dynamic change is validated by describing the solutions of previous research in terms of the model. The execution of change primitives (the implementation of change) and the techniques used to ensure the correct execution of the program (the management of change) are explained for each type of change supported by a particular solution.

1.4 Thesis Overview

Chapter 2 provides an overview of the previous research conducted in this field. Only a brief description of the solutions proposed by previous researchers is presented as the solutions are revisited in chapter 6. The model of a computer program is developed in chapter 3. Physical communication primitives are defined and the mapping of logical to physical communication primitives is described. Chapter 4 summarizes the implementation of change. This chapter ignores the management of change in order to identify the basic implementation functions. The management of change is described in chapter 5. This chapter presents a definition of a correctly executing program and outlines the options for ensuring that the implementation of change (i.e., the execution of change primitives) does not cause the program to fail. The theory of dynamic change developed in chapters 3, 4 and 5 is validated in chapter 6. Chapter 7 presents conclusions and future work.

Chapter 2

Previous Research

2.1 Changing Modules on the Fly

The earliest research in this area was conducted by Fabry [8]. His proposal requires an existing system which supports capability addressing [9]; i.e., one in which code segments are addressed indirectly. This system permits the replacement of *modules*, which are defined as 'the set of programs which implement all of the operations on a particular abstract data type.' A module is replaced by changing the capability address of a module. (The arrows in figure 2.1 represent capability addressing.)

A data locking mechanism (to synchronize updates) and an embedded version number (to prevent the utilization of an old data structure) are required for each instance of a data structure. Whenever a module is called, the data structure that it controls is locked and the version number is compared to an expected version number. If data structure's version number is not as expected, the module either updates the data structure via a conversion routine built into the module (if the version number is less than expected) or returns to calling module to be recalled (if the version number is greater than expected). In the latter case when the module is recalled the latest version of the module will be called.

Fabry's system does not support changes to the parameters of a module; i.e., the parameters of the call to the replacement module must be identical to the original module. The new version of the module must also contain a conversion routine to convert instances of the data structure that it controls.

Figure 2.1 Changing Modules on the Fly

2.2 Dynamic Alterable System (DAS)

The experimental operating system, DAS [12], provides a capability to update *modules* through the use of a special addressing scheme. Modules consist of code, object (shared data) and own (local data) segments. Each segment is addressed by a descriptor and the descriptors are linked together to define the module as shown in figure 2.2. A descriptor has four components: a kind which identifies the type of segment, a base which identifies the physical address of the segment, a link which points to the next descriptor in the chain, and a synchronization word which consists of a use count and a boolean flag to indicate that a module is being updated.

Modules are "replugged" by changing the links in the descriptor. Three replugin operations are provided: *chlink* to replace the code of a module containing object and/or own segments, *repcode* to replace the code of a module which does not contain object or own segments, and *repdat* to restructure data and replace code. *Chlink* and *repcode* are two step operations. The new segment is loaded into memory and the link field in a descriptor (an object or own descriptor in the case of *chlink* and a code descriptor for *repcode*) is overwritten. *Repdat* adds an additional step between the loading of segments and the overwriting of link fields. The additional step is the execution of a *data.restruct* procedure

which copies values from the old to the new data structure(s). As in the Fabry system, the interface of a module cannot be changed.

Figure 2.2 DAS Module

2.3 DMERT

The field administrative subsystem of the DMERT operating system [26] provides the capability to replace program functions in a telephone switch. The paper describes the implementation of dynamic change but does not address the issues of dynamic change management. The operating system uses indirect addressing for function calls. *Transfer vectors* point to the function's address. In order to replace a function, the new version is loaded into a special patch area and the transfer vector is overwritten to point to the new address.

2.4 Dynamic Module Replacement

Bloom [1] defines a system which supports the dynamic replacement of *guardians* in the Argus programming environment. Guardians contain a set of processes and a set of objects. Each guardian has an independent address space and communicates with other guardians via message passing. Changes are implemented to sub-systems, a group of one or more guardians. The interface to the sub-system cannot be changed. The guardians intended for update are halted and any ongoing transactions aborted. New instances of the guardians are created and initiated. State information is transferred from the old to new instances of guardians. The entire update transaction, although implemented as a number of separate commands, is treated as an atomic action.

2.5 DYMOS: Dynamic Modification System

DYMOS [4,18] is a complete dynamic modification system for the STARMOD language. It consists of a Command Interpreter, Source Code Manager, Editor, Compiler, Linker/Loader, and Run Time Support System. The system supports changes to *modules* and *procedures*. A module is composed of an export list, an import list, constant definitions, type definitions, variable declarations, procedure definitions, and optional initialization code. The start address of modules and procedures is stored in Module Address Tables and Procedure Address Tables. Each address in the tables includes a lock bit used to enforce mutual exclusion on processes that reference or modify an address entry. The first word of each procedure is also reserved. The first bit is a lock bit and the remainder of the word is used as a count of the number of active calls to the procedure. The execution of any procedure necessitates the locking of its entry in the Procedure Address Table while the procedure use count is incremented and decremented. Referencing global or shared variables also requires the locking of entries in the Module Address Table.

In order to initiate an update, the user specifies the modules and/or procedures to be updated, the ones to be deleted, the modules which must be inactive during update, and any time-out limitations. The format of the update command is

update <arg list1>[delete <arg list2>][when <arg list3>idle][within <limit>].

When the system receives an update command the first bit in the first word of those procedures specified in arg list3 is set. Provided that a time-out does not occur, when the use count of these procedures reaches zero, the procedures and modules specified in arg list1 and arg list2 are replaced and deleted as appropriate. The update and delete is executed as an indivisible operation.

DYMOS supports two strategies to update the interface of a module or procedure. If the interface to a procedure (module) is to be updated, all procedures (modules) affected by the change (i.e., the called and calling procedures (modules)) can be updated at the same time since DYMOS permits the addition, updating and deletion of several modules (procedures) as an atomic action. The alternative method is to include a conversion routine in the new procedure.

2.6 Object Oriented Languages

An object oriented approach to dynamic change management is presented by Stadel [25]. Stadel uses the object oriented concept of inheritance, the deferred class of the Eiffel language (abstract classes in Smalltalk and virtual classes in C++ are similar), and a dynamic link loader. The dynamic link loader, which maintains a global table of entry names and the corresponding load address, must be provided by the operating system. The dynamic link loader is called by a *configuration manager* which must be included in the program to be updated.

Only objects which are instances of a deferred class may be updated. The state of an object is preserved by *cloning*. Cloning may take place in the foreground (the execution of all other modules except the cloning module is halted) or in the background (other modules continue to execute), however background cloning is only applicable to container classes (a container class is one which does not call features of other objects passed as parameters to its routines or shared with other classes).

2.7 Ada

Burns & Wellington [3] explore Ada's support for dynamic change management. Three cases of Ada program component replacement are considered: components without explicit state information, passive components with state information, and active components with state information. In all cases the authors state that the interface to the component must remain unchanged.

The replacement of a component without state information is accomplished by placing a *hold* on calls to the original component, linking in the new component and removing the hold. The replacement of passive components with state information (i.e., packages) can be accommodated in one of two ways: by separating the code and state information into two load modules (thus reducing the problem to one of replacing a component without state information) or by storing the state information in another component or in a disk file which could then be read by the new component when the hold is removed. The same techniques for transferring state information are applicable during the replacement of active components (i.e., tasks), however the transfer of control is not as simple. The holding of calls to the original task may have to be selective, calls to certain entries are held while others are allowed to proceed. The authors state that general rules cannot be applied; each replacement must be treated as a special case.

An alternative solution [2] for a language based on Ada has also been proposed. Each package or task contains a delegation operation in which the new package or task is passed as a parameter to the old version. The state of the new component is gradually updated and the state of the old component gradually evolves to its initialization state; at which time it can be removed. This methodology allows for the new component to have more attributes than the old but not fewer. The major shortcoming of this approach is that each package or task must have an inherent conversion routine; thus, the restructuring is limited to those implementations which were anticipated in the initial design of the module.

2.9 PROTEL

The Procedure Oriented Type Enforcing Language (PROTEL) was designed at Bell-Northern Research for the development of software for large digital switching systems [10]. The basic building block in PROTEL is a *module*. A module consists of one or more separately compilable units called *sections*. Section may be one of two types: interface or implementation. Interface sections contain procedure headings for the operations provided by the module as well as type definitions for the procedure parameter types. Implementation sections contain the executable statements of the interface procedures along with additional type, data and procedure declarations.

PROTEL permits multiple instances of interface and implementation sections in a module. This feature allows new modules to be dynamically added to a running program and is commonly used to add options at a customer site. The modules contain entry procedures that are invoked automatically upon loading. These entry procedures announce the module's existence to the system and initialize *procedure variables* that point at its procedures. The new module can be called by the system through these procedure variables [4,17].

2.10 PODUS

Mark Segal and Ophir Frieder have collaborated on five papers [11, 21, 22, 23, 24] in the area of dynamic software updating. The Procedure-Oriented Dynamic Updating System (PODUS) is based on replacing individual *procedures* of a program by overwriting addresses in a *binding table*. The updating process is initiated by the user after the new version of the program has been loaded. The update command interrupts program

execution and examines the runtime stack. The contents of the stack are compared to a list of procedures in the program and the system then determines when each procedure can be updated. Procedures that have not changed are updated immediately (in fact unchanged procedures are not replaced, rather they are flagged as being modified) and all other procedures are updated only when they are not active. A procedure is defined as being active if and only if it is on the runtime stack or if its new version can directly or indirectly call a procedure that is already on the stack.

Interprocedures and *mapper procedures* are used to maintain consistency during updates. Mapper procedures map local static data from the old representation to the new representation. Similarly, interprocedures map the calling sequences and return codes of old procedures into the new procedures. These special procedures are required to allow a procedure that has not yet been updated to call another procedure that has already been updated.

2.11 Kramer and Magee

Kramer and Magee [14, 15, 16, 19] have developed a *configuration management system* which supports dynamic change management. The configuration management system is built into the CONIC operating system. The system to be updated is viewed by the configuration management system as a set of interconnected *nodes*. The configuration management system can load or delete nodes and establish or eliminate connections between nodes.

In order to effect changes, nodes are directed to a particular state. Three states are defined: *active*; *passive*; and *quiescent*. The active state is the normal state for the system and in this state a node can initiate, accept, and service transactions. After receiving a management request to enter the passive state, a node must complete all transactions that it initiated and refrain from initiating any new transactions. It may continue to accept and service transactions initiated by other nodes. A node is defined to be quiescent when it is in the passive state, it is not currently servicing a transaction, nor will it be required to service any future transactions. This state is achieved by directing all nodes connected to the node in question to the passive state. The quiescent node and the set of passive nodes connected to it are defined as the *passive set*.

In order to support systems with dependent transactions (transactions which require one or more consequent transactions), the passive set must be expanded to include

all nodes initiating transactions which result in subsequent transactions on the link or node targeted for change. To accomplish this an additional state is required, the general passive state. A node in the general passive state must continue to accept and service transactions but can only initiate consequent transactions. In bounded time all of the consequent transactions will be completed and all nodes will enter the more stringent passive state. Dependent transactions can also be accommodated by composition. Nodes with dependent transactions are grouped together to form a node at a higher level of abstraction. If composition can be applied to all nodes with dependent transactions the problem is reduced to an equivalent one with independent transactions. For a composite node to enter the passive state, all of its subnodes must be in the passive state.

2.12 Summary

The solutions to the implementation of dynamic change proposed by previous researchers cannot be extended to the general case. The proposed solutions only support certain types of changes and only within a specific environment. In order to develop a general solution, one which supports all types of changes in any environment, a generic environment must be defined and the types of changes classified. Options for implementing each type of change can then be explored. The design of a dynamic change management system would involve selecting an option for each type of change and mapping it to the specific environment.

Chapter 3

Model of a Computer Program

3.1 General

A program is a collection of *modules*. The program modules may be procedures in procedure-based language, tasks in a multi-tasking system, or some combination thereof. There is not necessarily a one-to-one mapping of source language modules to the model modules. Since the dynamic modification of a computer program requires the manipulation of the machine executable instructions and data corresponding to the program, module boundaries must be determined from examining machine code and not source language files. The optimization techniques of modern compilers (e.g., in-line procedures) can result in the merging of source language modules. Even in the case of non-optimizing compilers, several source language modules may be represented as a single module or a single source language module may be split into several modules.

For example, if a program consists of a number of tasks (processes) and a task contains a number of procedures, each procedure could be modeled as a separate module or several procedures and/or tasks could be modeled as a single module. In selecting a representation of a program, the decision on whether or not to combine modules is dependent upon the modification to be implemented. If the modification will only affect one procedure, then the program can be modeled as a collection of procedure modules, or even as one procedure module and a second module representing the rest of the program. If on the other hand, the modification involves the updating of several procedures in a particular task, it may be more practical to model the task as a single module. Thus, several representations of the same program are possible, or conversely, the same model

may represent two totally different programs; e.g., a module in figure 3.1 may represent a single procedure, a single task, several procedures, or several tasks.

Figure 3.1 Modules

The lines and arrows in figure 3.1 represent communication links. Normally, the modules that make up a program are interdependent; i.e., they exchange information with each other. Modules which do not communicate with other program modules, (a module which displays a clock for example) must communicate with the operating system or directly with a hardware component. If a particular module does not exchange any information with any other (program or operating system) module or hardware component, it cannot be contributing anything to the overall functionality of the program and could therefore be eliminated from the program.

3.2 Modules

A module must have some type of input and output in order to facilitate communications. When a module receives input it performs a transformation and produces output. This description of a module is also a description of a state machine. A state machine is a black box with input(s), output(s), a *state transition function*, and an *output transition function*. The module (state machine) communicates with other modules through its inputs and outputs. The state transition function determines how the module reacts to various inputs and the output transition function determines the module's output given a particular input and current state. The collection of inputs to which a module responds is known as its *input alphabet*. The term *output alphabet* is used to describe the

collection of possible outputs from the module. Since the modules of a computer program can be viewed as state machines, a computer program can be modeled as a collection of communicating state machines.

Figure 3.2 Module

State machines can be represented in one of two ways [13]; as a *state-assigned machine* or as a *transition-assigned machine*. The main difference between the two representations is that in a state-assigned machine output is associated with the state whereas a transition-assigned machine associates the output with the transitions between states. In the program model, a state-assigned machine representation has been used. Thus, a module is defined, at time n , in terms of its input, $I(n)$, its output, $O(n)$, and its current state $S(n)$ as shown in figure 3.2. The input to a module is either the output of some other module or a signal from the hardware/operating system. These inputs and the module's current state are the independent variables of the state transition function, $S(n+1) = \sigma\{S(n), I(n)\}$, and thus determine the module's next state. The output of a module is a function of its current state; $O(n) = \omega\{S(n)\}$. The state transition and output functions are encompassed in the code that a module executes.

An extended state machine description of the state of a module is used in the program model. In an extended state machine, the state consists of two components: the *control state* and the *data state*. The control state defines what the module is doing, and the data state is a collection of data variables. The data variables are local to the module; they cannot be directly accessed by any other module. A module's control state is classified with respect to the module's interaction with other modules. A module may be receiving information from another module, sending information to another module, or doing something else, such as modifying the values of its state variables, which does not

involve communications with other modules. In other words, a module may be in a *send state*, a *receive state*, or a *processing state*. A module's control state is determined by examining intermodule communications.

The ability to define the control state of a module in terms of intermodule communications was the key factor in selecting a state-assigned machine representation for a computer module and also for not including the current input as a dependent variable of the output function.

3.3 Intermodule Communications

In order for the output of one module to become the input of another module there must be some type of communications link between the two modules. In the model, the communications link is a direct connection between two modules and the information is transferred over the link via message passing. Since a link only connects two modules, a module must have multiple inputs and outputs if it is to have more than one communications partner. In fact, a pair of communication links exist between each and every module, one for incoming messages and one for outgoing messages, however, in practice a module will use only a limited number of communication links as a given module only has a limited number of communication partners. The selection of the link over which a particular output message is transmitted is state dependent as the only independent variable in the output function is the current state.

3.3.1 Communications Primitives

The transmission and reception of messages requires a set of communication primitives. Three communication primitives³ have been defined: a *non-blocking send* in which the recipient of the message is explicitly named; a *blocking receive-any*; and, a *blocking receive-specific*. There is a one-to-one mapping of control states and communication primitives: a send primitive is executed in a send state; a receive-any in a receive-any state; and a receive-specific in a receive-specific state. When a module enters a

³ These particular communication primitives (as opposed to a blocking send or a non-blocking receive) were chosen to facilitate the mapping of logical communication primitives to the model primitives (described in section 3.3.2). For example, it is relatively easy to model a logical blocking send as a non-blocking send followed by a blocking receive, however, modeling a logical non-blocking send with a physical blocking send is much more complicated.

send state, an output message is transmitted over one of the module's outgoing links. The selection of which link to use is synonymous with the destination of the message as every possible destination has its own link. The destination of, and the actual message to be transmitted, may be stored in data variables or they may be hard-coded in the output function. Once the message is transmitted, the originator module will proceed to its next state. The next state may be another send state, one of the receive states or a processing state. In cases where the same output must be transmitted to several modules, a separate message, each originating from a separate state, is sent to each module.

When a module enters a receive state it will remain in that state until it receives a message. The only difference between a receive-any and a receive-specific state is the selection of links from which the module will accept input. In a receive-any state, the module will proceed to its next state upon the receipt of a message from any other module, whereas in a receive-specific state the module will accept messages only from a specified module or group of modules.

These communications primitives do not guarantee the delivery of messages. If the originating module enters a send state before the recipient module enters a receive state, the message will be transmitted but not delivered. In addition, the originator module is not provided with any indication that its message has not been delivered; it proceeds to its next state as soon as the message is transmitted.

3.3.2 Logical and Physical Communication Models

These communication primitives describe the *physical*, as opposed to the *logical*, exchange of information between modules. The logical exchange of information between program modules uses communications primitives provided by the operating system and programming language. The logical communication primitives must be mapped to the model's physical communication primitives. A logical exchange of information may take place between modules within the same process (intraprocess communications), or modules in separate processes (interprocess communications).

3.3.2.1 Interprocess Communications

In interprocess communications, a direct communication link between program modules does not exist, modules communicate indirectly via the operating system. The simplest type of interprocess communications is via shared memory. Modules exchange

information by reading and writing to the shared memory. An *abstract* module can be defined which encapsulates the memory; the data variables of this abstract module are the actual memory locations and the contents of the memory locations are the value of the variables. The state transition function of an abstract module is defined by the pseudo-code of figure 3.3.

Writing to a memory location involves the transmission of a message to the abstract module directing it to update one of its variables. Similarly, reading a memory location can be viewed as two message exercise; one message transmitted to the abstract module requesting the current value of a variable, and a second message transmitted to the requesting module containing the value of the variable. The requesting module, after transmitting the "read" message, enters a receive-specific state to await the message containing the value of the variable. The message transmitted to an abstract module consist of three components: the id of the sender; the message type (either read or write); and the text of the message. The text of the message contains the new value of the data variable in the case of a write message. In the case of a read message, the abstract module sets the message text to the value of the data variable and sends a reply message.

```

Abstract Memory Module
  variables static variable, sender, msg_type, msg_text
  Begin Loop
    recv-any (sender, msg_type, msg_text)
    if msg_type = read
      msg_text = variable
      send (sender, msg_type, msg_text)
    else if msg_type = write
      variable = msg_text
    end if
  End Loop

```

Figure 3.3 Abstract Memory Module

The other type of interprocess communications is message passing. In a message passing system, two communicating program modules also exchange information by sending messages to a third module, but in this case the third module is not an abstract module; it is an operating system module, the *message handler*. The transmission of a single logical message from an originator to a recipient module requires a physical transmission of a minimum of three messages: one from the originator to the message handler; a second from the recipient module to the message handler; and a third from the message handler to the recipient. The total number of messages actually transmitted is dependent upon the logical message passing primitives provided by the operating system.

All logical message passing primitives executed by a program module involve the execution of a physical send primitive: the program module must send a message to the message handler. Most logical primitives also require the execution of the receive-specific primitive, the only exception being a logical non-blocking send. The actual number of messages transmitted between modules is dependent upon the communications paradigm. For example, a *rendezvous* (depicted in figure 3.5) requires the execution of five physical primitives and thus the transmission of five messages. In a rendezvous, the originator executes a logical blocking send and the recipient executes a logical blocking receive followed by a logical non-blocking send (reply). In terms of physical primitives, the originator executes a send followed by a receive-specific. The recipient executes a send, a receive-specific, and a second send.

Figure 3.4 Communications with Abstract Module

Figure 3.5 Rendezvous

The message handler can be composed of several modules; i.e., the originator and recipient modules may send messages to different modules of the operating system. An operating system module, upon receipt of a message from an originator module, forwards the message, or the location of the message, to another operating system module with which the recipient module communicates. The first module may forward the logical

message directly or indirectly via additional intermediate modules. In the case of distributed systems, these modules may be in different machines. The actual composition of the message handler, a single module or a collection of two or more modules, does not affect the logical transmission of modules and therefore, for the sake of simplicity, all operating system modules involved in the logical transfer of a message are represented as a single module in the model; i.e., in the case of a message handler composed of several modules, all the message handler modules are combined to form a single module.

Messages sent to the message handler have three components; a message type which corresponds to the logical primitive to be executed, the logical destination (or sources to accept messages from in the case of logical receive primitives), and the actual message or location of the message. The message type is the real destination of the message, it determines which sub-module of message handler actually receives the message. The logical destination determines where the message or location of the message is stored until it is delivered to the recipient module.

3.3.2.2 Intraprocess Communications

There are two basic forms of intraprocess communications: (1) the use of global memory, either simple variables or data structures; and (2) procedure calls or jumps. Global memory is treated the same as shared memory; i.e., an abstract module is defined which encapsulates the memory and the program modules read and write to the memory by sending and receiving messages.

Procedure calls and jumps are the only form of logical communications which do not require an intermediate module between the two communicating program modules. A jump is modeled as the execution of a single send by the originator and a receive by the recipient module. A procedure call is slightly more involved. The initial state of a procedure module is receive-any. A procedure is called by sending it a message, the content⁴ of the message being the parameters of the procedure. A procedure returns by sending a message. The calling procedure, after executing the send which initiated the called procedure, enters a receive-specific state in which it will receive the return message.

⁴ A procedure call can also be modeled by having the calling module write the parameters in global memory, for example on the stack, and then send a message to the procedure. In this case, the content of the message would be the location of the parameters.

In systems that use direct or relative addressing for procedure calls, the logical communication link is identical to the physical link. However, in systems using indirect procedure calls, the logical link is modeled as two physical links and an intermediate *procedure handler module* as illustrated in figures 3.6 and 3.7.

Figure 3.6 Indirect Procedure Calls

Module	Logical	Physical
Calling Procedure	return code = call (procedure, parameters)	send to proc-handler (my_id, proc_id, parameters) receive from procedure (return_code)
Called Procedure	procedure (parameters) ... return (return_code)	receive-any (caller, parameters) ... send to caller (return_code)
Procedure Handler	jump absolute	receive-any (source, destination, msg) send to destination (source, msg)

Figure 3.7 Indirect Procedure Calls - Pseudo Code

In this case, the calling module sends a message, consisting of a destination and the parameters of the procedure, to the procedure handler module. The procedure handler, a

module of the operating system, looks up the destination in a table and forwards the parameter portion of the message and the address of the caller to the procedure module. The called procedure module sends the return message directly to the calling module.

3.3.2.3 Logical and Physical Representations of Communication Links

The previous two sections have demonstrated that the mapping of a logical exchange of information between two program modules to the physical communication primitives of the model normally requires an intermediate communication handler module as illustrated in figure 3.7; the exceptions being direct procedure calls and jumps. The communications handler module may be an abstract memory module, a message handler, or an indirect procedure lookup table.

Figure 3.7 Communication Links

Thus, two models of a program are possible: the logical model and the physical model. The logical model does not include those modules of the operating system (message and procedure handlers) that transport messages. The only operating system modules included in a logical model are those modules, such as device drivers, semaphores and shared libraries, that enable the program to interact with its environment. Programs which include built in device drivers communicate directly with hardware components. In this case each hardware device is encapsulated by an abstract module and these abstract modules are included in the module. On the other hand, the physical representation of a

computer program includes those operating system modules which perform the actual transport of logical messages. The choice of whether to represent the interconnections between modules as logical links or as physical links is dependent upon the stage of the modification process. As is demonstrated in the following two chapters, a logical representation is required during the analysis phase (i.e., determining what has to be done to implement the modification) whereas a physical representation is required during the implementation phase.

3.3.3 Multiple Instances of Modules

The description of intermodule communications thus far has not addressed the issue of ensuring the delivery of messages. For example, a procedure may be called at any time and may in fact call itself in the case of recursive procedures. However, once a procedure module has received a message it will not reenter its initial receive state (and thus be in a position to respond to a second call) until the first call is complete. One solution is to have multiple instances of the procedure module with one of the instances always in its initial state. Multiple instances of the same module execute the same code, but each has a separate set of local data variables. Static variables are modeled as abstract memory modules as they can be accessed by more than one instance of the module.

This solution has been adopted for the program model and is implemented by creating a clone of the procedure module in conjunction with the receipt of the call message. In order to eliminate an unbounded number of instances of a module, the first instance of the procedure module can be deleted in conjunction with the transmission of the return message. Procedure modules, upon leaving a receive-any state, create a second instance of themselves and delete themselves after transmitting a return message. The presence of multiple instances ensures that a procedure module is always in a receive state prior to the transmission of a message by the originator.

3.4 Operating System and Meta-Level Modules

An operating system is more than a transport mechanism which enables program modules to communicate with one another and with hardware peripherals. An operating system also performs operations such as loading programs into memory, memory management, and task switching. In performing these operations, the operating system treats the program modules as data, thus the modules of the operating system which

perform these functions are *meta-level modules*. The other operating system modules are classified as *object level modules* as their interaction with the program occurs at the same level the interaction between program modules. A meta-level module has all the properties of an object level module but has an expanded data state space. The data variables of a meta-level module may include the data variables and code of an object level module. The restriction that data variables are not accessible to any other module is not applicable to modules at a higher level, only to modules at the same level.

The program itself may have meta-level modules. For example, a program may have a module which verifies data integrity or which monitors the rest of the program for failures. These types of modules require access to the data state of other modules and therefore must be classified as meta-level modules. However, it is possible to model a program containing these types of modules as a collection of object level modules, with no meta-level modules. If those portions of the data state accessed by more than one module are modeled as separate global or shared memory modules, then access to the 'data state' of another module is via normal communication links and thus all modules exist at the object level. A program may also have modules which must access the state transition and output functions of other modules, e.g., a program with a dynamic loader. This type of module treats the code of another module as data. In this case, module boundaries can not be restructured such that all modules exist at the object level.

3.5 Summary of the Program Model

A computer program is a collection of communicating modules, where the modules are represented as extended state machines and the communications between modules occurs via message passing. A module is completely defined by its code (state transition and output functions) and current state. An abstract module's code is not accessible to any other module. A module's data state is a collection of variables and its control state class is defined in terms of its communication activity. The collection of modules is not limited to program modules; parts of the operating system, and in some cases parts of the hardware, are also included as modules in the model. Modules may be grouped together or a module may be split into two or more separate modules, and thus several representations of the same computer program are possible. The decision whether to represent a given module as a separate module or to combine it with another module is dependent upon the changes to be made to the program.

Notwithstanding the multiple representations which are possible due to combining and separating modules, two representations of the program are required for dynamic change management purposes; one in which the interconnections between modules are represented as logical links and one in which the logical links are mapped to physical links. The mapping of logical links to physical links normally requires intermediate operating system module(s) which acts as a transport mechanism for the messages. A logical model is used during the analysis of change and the physical model during the implementation of change.

Chapter 4

The Implementation of Change

4.1 Modifications

A computer program has been modeled as a collection of communicating modules. Given this model, any modification to a computer program, no matter how complex, can be broken down into a number of smaller changes to the individual program modules and the interconnections between modules. Modules may be added to or deleted from the program, or existing modules may be updated.

With respect to the interconnections between modules, the possible changes are dependent upon whether the interconnections represent physical or logical links. Logical links may be added or deleted, but physical links cannot be changed; a physical communications link always exists, the only question is whether or not it is used by the program. Thus, in order to identify what must be changed, the interconnections between modules must represent logical communication links.

The changes to the individual modules and the logical interconnections between modules required to implement a particular modification are dependent upon how the modules of the program have been modeled. For example, a procedure may be modeled as a separate module or as part of a task module. If the modification involves the deletion of that procedure, the changes include the deletion of a module (procedure) in the first case, whereas in the second case, they include the update of a module (task).

All logical links can be mapped to a set of physical links.⁵ If the entire set of physical links used by a program is referred to as a *configuration of communication links*, the addition or deletion of a logical communication link is a *reconfiguration of communication links*.

The changes which can be made to a computer program (adding and deleting modules, updating a module's code or current state, and the reconfiguration of communication links) are the same whether the program is modified dynamically or statically. The difference between a dynamic modification scheme and a static one is not in what can be modified, rather in how and when the modification is implemented.

4.2 Implementation of Modifications

A modification of a computer program is implemented as a series of changes to the individual modules of the program. Each change in this series is a meta-level operation on the program. In order to implement the *modification operations* a meta-level entity is required. In a dynamic modification scheme, this entity is the *change management system*. The implementation of modification operations by the change management system is analogous to the editing of source language files in a static modification scheme, bearing in mind that the modules of the model do not necessarily represent a one-to-one mapping of source language modules. Just as an editor manipulates the characters representing a module, the change management system "edits" the memory locations corresponding to the module's code (state transition and output functions) and state variables. An editor implements modification operations by executing a series of basic functions such as overwriting a character, cutting a block of characters, and pasting a block of characters. Similar functions are executed by the change management system. These functions are referred to as *change primitives*.

There is, however, one important difference between the dynamic modification of a computer program and the static editing of source language files. In a static modification scheme, a modification operation (editing a module) does not affect the execution of the program until the new version of the program is compiled, loaded and begins execution. Although change primitives are executed in series, they all take effect at the same time and

⁵ In the case of direct procedure calls or jumps, each logical link maps to a single physical link.

with all (unchanged, updated and new) modules, except *persistent data modules*,⁶ in their initial state. In a dynamic modification scheme each change primitive takes effect as it is executed and therefore the dynamic implementation of a modification operation can introduce incompatibilities among the modules which make up the program which in turn can result in the incorrect execution of the program. Therefore, as well as executing change primitives, the change management system must also conduct other operations to ensure the correct execution of the program.

In order to identify change primitives, modification operations must be isolated from those operations intended to ensure the correct execution of the program. The additional operations which must be performed to ensure the correct execution of the program can be ignored if it is assumed that: (1) the program does not contain any persistent data variables (the modification of programs with persistent data variables is examined in Chapter 5); and, (2) the same conditions hold as in the case of a static modification; namely, all change primitives take effect at the same point in time, and all modules are in their initial state at that point in time. In other words, it is assumed that the change management system can halt the execution of the program at a point in time when all modules are in their initial state, implement the modification operations, and restart the program. With these assumptions, the only activity performed by the change management system during the modification is the execution of change primitives. While it is impractical for a change management system to be able to halt a program in its initial state, this assumption permits the isolation of change implementation from change management and allows an implementation strategy for each type of modification operation to be identified. This assumption is applicable only in this chapter. In subsequent chapters, the assumption has been relaxed.

4.2.1 Module Updates

Determining how to implement module update operations requires an examination of how an operating system keeps track of a module. A module occupies two⁷ sections of

⁶ A persistent data variable is one in which the value of the variable must be maintained between invocations of the program, i.e. it must be stored in non-volatile memory. If a program with persistent data is halted and restarted, it does not resume execution its initial state.

⁷ An abstract module only occupies one section of memory as it does not have to store the code that it executes.

memory (which may or may not be contiguous), one for its code and one for its state variables. The operating system must also keep track of the control state of a module, however, the control state of a module can be ignored as it has been assumed that all modules are in their initial state when the modification takes effect, and therefore, the control state of a module can be determined from examining the initial instruction in its code section. Thus, given the above assumptions, updating the control state of a module requires an update of the module's code.⁸ Since a module is simply a block of memory locations, the update of a module by the change management system is similar to replacing a block of text by an editor. Just as an editor can replace a block of text by cutting and pasting or by overwriting characters, the change management system can implement the operation of updating a module by deleting it (cutting) and adding (pasting) an updated version, or by directly overwriting the memory locations occupied by the code and data state variables. In order to distinguish between these two implementations, the operation of updating a module has been sub-divided into two operations; *module replacement* and *module overwriting*.

4.2.1.1 Module Replacement

A module replacement operation is implemented by: (1) loading a new version of the module (adding a module) initialized to the desired state; (2) redirecting the message traffic intended for the old module to the new module (reconfiguration of communication links); and (3) deleting the original module. Thus, although module replacement has been defined as a distinct operation, it is implemented as a combination of three separate modification operations (these operations may be performed in any order). The change primitives that must be executed to implement a module replacement operation are the same as those for the separate operations. Since modules may be combined or separated, module replacement also refers to the replacement of a single module with multiple modules and the replacement of multiple modules with a single module.

4.2.1.2 Module Overwriting

A module may also be updated by overwriting the memory locations occupied by its code and data variables. The overwriting of a single memory location is a change

⁸ Control state updates are examined in greater detail in section 4.2.4.

primitive; it is similar to the overwriting of a character by an editor. If the module overwrite operation involves a change to more than one memory location, it is implemented by executing a series of change primitives, each change primitive being a memory overwrite.

Given the earlier assumptions, the memory overwrite primitive must be executed by a meta-level module. However, without these assumptions, the fact that the module to be updated also has access to these memory locations can be exploited in a dynamic modification scheme. In the course of executing, a module constantly updates its data variables, and, in the case of self-modifying code, its state transition and output functions. Therefore, the change management system could send the module a message, assuming that the content and format of the message is recognized (i.e., it is contained in the input alphabet), directing a module to update the value of data variables or to begin its self-modification process. In order to use this technique, the module to be updated must have been designed with a dynamic modification capability.

Although a module with self-modifying code is able to update its state transition and output functions, it does so by updating the data variables of another module. A module with self-modifying code is modeled as two modules, a program module and an abstract meta-level memory module. When a module overwrites a memory location containing part of its code, it is, in effect, sending a message to the abstract module requesting it to update one of its data variables.

All memory locations which can be overwritten are data variables (of either an object level module or a meta-level abstract module) and therefore the memory overwrite change primitive is more accurately described as *overwriting a data variable*. This change primitive may be executed by a meta-level module or an object level module, however, the control of the execution is determined by the change management system. In other words, the change management system may execute the change primitive itself or it may delegate the execution to a program or operating system module.

4.2.2 Addition and Deletion of Modules

The assumption that all modification operations take effect at the same point in time also permits the interdependence of modification operations to be ignored. A modification to a program rarely consists of a single operation on a single module and, in the case of the deletion of module, the modification must include update operations on all

of the communications partners of the module being deleted. The update operations to the communications partners may just involve the reconfiguration of communication links or they may be more involved. In fact, a module may be effectively deleted by reconfiguring communication links. If a module cannot communicate with other modules, it cannot affect⁹ the execution of the program. Thus, assuming that the module will be effectively deleted during the course of implementing other modification operations, the operation of deleting a module is a garbage collection process. The module's code and data variables must be removed from memory, and in certain cases, data structures maintained by the operating system, such as process control blocks and message queues, may have to be cleared. Depending upon operating system requirements, the module may not be actually removed from memory, the memory previously allocated to the module could just be identified as available. This garbage collection process is a change primitive. The operation of deleting a module requires the implementation of other modification operations (such as the reconfiguration of communication links) to effectively delete the module and the execution of a single change primitive, *removing a module*, to remove the module from memory and clear data structures.

The addition of a module is the opposite of deleting a module. The module's code and data variables must be loaded into memory and operating system data structures have to be updated. Link editing (only within the new module, not within other program modules) of procedure call addresses and global or shared memory segments is also required, as is the provision of process names to facilitate message passing. Collectively, these activities are a change primitive and are referred to as *loading a module*.

As in the case of overwriting a data variable, the change management system may execute the change primitives, removing a module or loading a module, associated with the modification operations of adding or deleting a module directly, or it may delegate the execution to another module by sending the other module a message.

The change primitives, removing a module and loading a module, could be divided into series of smaller operations, however, the execution of either of these change primitives cannot affect the execution of the program. Before removing a module it must

⁹ A module which has been effectively deleted, once the program is restarted, can continue to execute and therefore slow down the execution of other modules, but it cannot affect the output produced by any other module.

be effectively deleted from the program and, as is demonstrated in chapter 5, the loading of a module also requires the implementation of other modification operations before the new module will effect the execution of the program. Thus, there is no benefit in subdividing these change primitives.

4.2.3 Reconfiguration of Communication Links

The reconfiguration of communication links is not a change in itself, rather it is the result of a change to a module. A logical link is synonymous with the destination of a message; each destination has its own link. The destination may be hard-coded into the output function or stored in a state variable. Thus, in order to delete or add a logical link, a module's state or output function must be updated. (The state of a module may be indirectly updated by updating its state transition function such that the module will proceed to a state with the desired output.)

The addition or deletion of a logical link is the result of changing the logical destination of a message, but a change to the logical destination of a message does not necessarily result in the reconfiguration of communication links. For example, consider a situation involving three modules and three messages. Before the modification, Module A sends messages 1 and 2 to Module B and message 3 to Module C. After the modification message 2 is sent to module C. In this example, the logical destination of a message was changed, however, no logical links were deleted nor were any created. Therefore, this operation is just a module update operation, not a reconfiguration of communication links.

A change to the destination of messages is one type of output change, the format or content of a message could also be updated. These output changes are also the result of updating a module's code or state. Similarly, changes to the input of a module are a result of updating the output of some other module. Thus, although a module is represented as an extended state machine and thus characterized by its input, output, state transition function, output function and current state, the only components of a module which can be directly updated are its code (state transition and output function) and current state.

Although the reconfiguration of communication links is not an operation in itself, but rather the result of a module update operation, communication link operations are special in two respects: (1) communication link operations may be implemented by updating operating system modules; and (2) communication link operations must be implemented by overwriting memory locations.

In a static modification scheme, there are only two methods of reconfiguring a logical communications link: (1) by updating the originator module(s), or (2) by updating the recipient module to redirect the message to another module. Consider the replacement of a procedure module B by module C in figures 4.1 and 4.2. In figure 4.1, the calling module, A, is updated to call module C vice module B, thus module C replaces module B. If more than one module calls module B, all the calling modules would have to be updated to call the replacement module C.

Figure 4.1 Updating the Originator

In figure 4.2, the called module, B, is updated to call module C. Thus, the direct communications link from module A to B is reconfigured to an indirect link from A to B to C. A variation of this technique would have module B executing a jump to module C. In this case, the return link between C and A would be direct. In either case, the modification operation of reconfiguring communication links to the module being replaced is implemented by performing a second update on module B. This method requires fewer modification operations than updating all the originator modules in situations where the module to be replaced has several communication partners.

Figure 4.2 Updating the Recipient

These two options for reconfiguring communication links (updating all of the communication partners or performing a second update on the original version of the module to be replaced) can also be used in a dynamic modification scheme, however, in a dynamic modification scheme a third option is available; updating physical links. Except in the case of direct procedure calls or jumps, a logical link is mapped into two physical links connected by an intermediate operating system module. The output of the intermediate module is governed by the same rules applicable to any other module; i.e., $O(n) = \omega(S(n))$. Thus, the logical destination of a message may be updated by modifying the code or state of the intermediate module which is part of the physical link between the two communicating program modules.

The reconfiguration of a communication link is the result of a module update operations, however, such an update must be implemented by direct overwriting of data variables, not via module replacement. Module replacement can create a logical link but a logical link cannot be deleted as the result of a module replacement operation. Consider a simple program consisting of two procedure modules, A and B, and a modification

involving the update of the called procedure, module B. Module B may be updated by replacing it; i.e., loading a new module, removing the original and reconfiguring the communication links between A and B. The reconfiguration of the communication link between modules A and B will require the update of module A. If this second update is also implemented via module replacement, it will result in the establishment of the logical link between the new modules, but it will not result in the deletion of the logical link between the original modules. Since the logical link has not been deleted, it will be impossible to remove either of the original modules as neither has been effectively deleted from the program. This situation is depicted in figure 4.3.

Figure 4.3 Reconfiguration of Communication Links by Module Replacement

The overwriting of a data variable that contains the logical destination of a message has the effect deleting the old link and establishing the new link simultaneously. Once the memory location containing the variable is overwritten, all future message will be directed over the new link and the original link will never be used again. This situation is depicted in figure 4.4. The data variable in Module A containing the destination of the procedure call (Module B) is overwritten with the destination of the replacement module (Module B*). The return link from module B to A does not have to be reconfigured; it will be removed at the same time module B is deleted. The reconfiguration of the incoming link effectively deleted Module B from the program. Module B will eventually enter a receive state, and, since no other module will send it a message, it will never leave

this state. The link from Module B* to A is established when Module B* is loaded into memory.

Figure 4.4 Reconfiguration of Communication Links by Overwriting

The reconfiguration of communication links is the result of a module update operation. This particular module update operation cannot be implemented by module replacement; it must be implemented by overwriting a data variable.

4.2.4 Control State Updates

In a static modification scheme, the control state of a module can be updated only indirectly by updating its state transition function. The control state is a description of what a module is doing; i.e., an identification of which portion of its code it is executing or will be executing in the case of a module whose execution has been suspended¹⁰ by the operating system. In a static modification scheme, all modules are in their initial state and thus, the portion of code which will be executed is the initial instruction in the module's code section. However, in a dynamic modification scheme, a second option is available, but only in the case of task modules.

¹⁰ The suspension of a module's execution is a meta level operation performed by the operating system.

The operating system maintains the control state of each module of a program. The manner in which this is accomplished is dependent upon the type of module, the state of the module, and whether the module is executing or is suspended. In the case of task modules, the identifier of the code to be executed by the module is either stored in the CPU's program counter or in a data structure; e.g., a process control block. It is assumed that the value of the program counter cannot be changed except under program control, thus the control state of an executing task module cannot be updated. A suspended task module's control state can be updated by changing the program counter value in the process control block. A control state change implemented in this manner can also result in an update of the module's state transition and output functions if the new value of the program counter corresponds to a memory location outside the limits defined for the module's code section.

This technique is not applicable to procedure modules. An operating system does not maintain a data structure, such as a process control block, for individual procedures. A procedure module is a sub-module of a task. If a procedure is the active sub-module of its parent task module, its control state is synonymous with the task module's control state. If it is not active, it must be in a receive state; it has either called another procedure and is waiting for the return message (a receive-specific state); or it is waiting to be called (a receive-any state). If the procedure module is in any other state it must be the active sub-module.

The value of the next instruction to be executed by an inactive procedure can be determined (the return address of a procedure which has called another procedure is normally stored on the stack and the address containing the initial instruction of a procedure which has not been called is normally stored in a data variable of the calling procedure or an intermediate procedure handling module), however, overwriting the location containing this value will not result in an update to the module's control state. The procedure module will still be in a receive state. The actual effect of overwriting one of these is a reconfiguration of communication links. The overwriting of a procedure return address is explored in greater detail in Appendix A.

Thus, the control state of a module can only be updated by updating the module's state transition function, except in the case of a suspended task module, in which case the control state may also be updated by overwriting the contents of the program counter variable in an operating system data structure.

4.3 Implementation of Change - Summary

A modification to a computer program can be broken down into a collection of smaller changes to the individual modules which make up the program and the communication links between these modules. Each change in the collection is meta-level operation and is implemented by executing one or more change primitives. The change primitives are: loading a module, removing a module, and overwriting a data variable. A change primitive may be executed directly by the change management system or the change management system may delegate the execution to another meta-level module or to an object level module. The delegation is accomplished via message passing.

Modification operations can be described in terms of other modification operations; e.g., the reconfiguration of communication links is an update to a module and the update to a module may be implemented by adding a module, deleting a module and reconfiguring communication links. Thus a one-to-one mapping of modification operations to change primitives is not possible. The identification of which change primitives to execute in order to implement a particular modification operation is one aspect of the change management process.

Chapter 5

Management of Change

5.1 Introduction

Given a set of modification operations to implement, the change management system must identify the set of change primitives to be executed, determine the order in which to execute them, and then cause them to be executed. During the performance of these activities the change management system is constrained by the requirement that the program continues to execute correctly.¹¹ Given that the execution of a valid, ordered set of change primitives is a straightforward exercise, the management of change is a three step process: (1) the generation of a set of change primitives; (2) the ordering of the set of change primitives; and (3) the validation of the ordered set. This chapter examines each step in this process, after first defining the notion of a "correctly executing program".

5.2 Correctly Executing Program

A correctly executing program is one which satisfies its functional specification. Every computer program has a functional specification; it may be explicit, implicit, or parts may be explicit and parts may be implicit. In those cases where a program's execution is not consistent with its functional specification (i.e., a program with a bug), an implicit functional specification which it does satisfy is assumed.

¹¹ It has been assumed that the modification has been designed correctly and therefore, after all modification operations have been implemented, the modified program will execute correctly.

The modification of a computer program is a maintenance activity. Software maintenance activities have been classified into four categories [20] based upon the reason for the change: *corrective maintenance* is the removal of errors; the addition of new capabilities and improvements to existing capabilities is referred to as *perfective maintenance*; *adaptive maintenance* refers to modifying software to properly interface with a changing environment; and *preventative maintenance* involves updating a program to improve future maintainability. It is possible for a modification to include more than one type of maintenance activity. Regardless of the reason(s) for the modification, modifications can be divided into two classes: those in which the original and modified programs satisfy different functional specifications (corrective maintenance and the majority of perfective maintenance), and those in which the original and modified programs satisfy the same functional specification (preventative, adaptive, and some perfective maintenance). Nevertheless, the specification satisfied by the modified program will be referred to as a *revised specification* regardless of whether or not it is identical to the *original specification*.

In a static modification scheme, all change primitives take effect at the same point in time and therefore the application always satisfies either the original or the revised specification. However, if the program is modified dynamically, each change primitive takes effect the instant it is executed. After the execution of a single change primitive, the program will satisfy a new specification, which may or may not be identical to the original specification. This continues until all the required modification operations have been implemented, at which point the program satisfies the revised specification. In other words, during the dynamic implementation of a modification, the program satisfies a series of *interim specifications* culminating in the revised specification. An interim specification may be identical to either the original or the revised specification or it may contain elements from both. Therefore, it is possible that, at certain points during the dynamic implementation of a modification, the program does not completely satisfy either the original or revised specification. This does not, however, imply that the program is executing incorrectly, but it does require a different definition of a correctly executing program.

5.2.1 Consistency

A computer program is a collection of communicating modules. If program is executing correctly, it will continue to execute correctly unless it encounters an

inconsistency. An inconsistency may exist in: (1) the code of a module; (2) the transmission of messages among modules; or (3) in the data state of the program. Thus, during the implementation of a modification, the program will continue to execute correctly unless the implementation of the modification introduces an inconsistency. The introduction of an inconsistency will not result in the immediate failure of the program. The program as a whole will continue to execute correctly until such time as: (1) a module with an inconsistency in its code executes that code; (2) an inconsistent message transmission is attempted; or (3) an inconsistent data variable is accessed. Thus, there are two approaches which can be taken to secure the correct execution of a program during the implementation of a modification. With the execution of each change primitive, the change management system can either: (1) ensure that the execution of the change primitive does not introduce any inconsistencies; or (2) if inconsistencies are introduced, prevent the execution of those portions of code which would reveal the inconsistency. The first approach is referred to as *maintaining program consistency* and the second as *execution control*. Maintaining program consistency is more elegant, however execution control may be more practical. In fact the two approaches may be mixed: e.g., the consistency of code and data could be maintained while inconsistencies in the transmission of messages could be handled via execution control.

5.3 The Change Management Process

The input to the change management process is a set of modification operations. Figure 5.1 depicts the logical models of an arbitrary program before and after a modification. The modification operations in this example are:

add new module F;

delete module C;

update module B;

add links B-A, D-F, F-E; and

delete links A-B, A-C, E-C.

Figure 5.1 Modification

The translation of these modification operations into a set of change primitives and determining the order in which to execute them (the change management process) is dependent upon the method of dealing with potential inconsistencies.

5.4 Management of Code Consistency

Given that the reconfiguration of communication links must be implemented by the execution of variable overwrite primitives, every modification operation can be described in terms of change primitives. The deletion of a module requires the execution of a module removal primitive and one or more memory overwrite primitives (to reconfigure communication links); the addition of a module requires the execution of the load module primitive and one or more memory overwrite primitives; and, the replacement of a module requires the execution of all three types of change primitives.

Since there are two options for the implementation of module update operations (except for those module update operations which are communication link operations), several different sets of change primitives can be generated. The choice of implementation method is dependent on how inconsistencies are to be handled. If code consistency is to be maintained, module updates involving changes to a module's code must be implemented

via module replacement whereas execution control is the required change management technique for module overwrites.

Updating a module by direct overwriting involves the execution of several variable overwrite primitives (except in the rare instance that an update to module's code can be accomplished by overwriting one memory location), one for each memory location to be updated. A module which is updated in this manner will have inconsistencies in its state transition and/or output function until the execution of the last change primitive. If this option is chosen for implementing module update operations, the output of this step in the change management process not only includes a set of change primitives but also a set of execution control conditions; i.e., a set of memory locations which cannot be accessed during the execution of a corresponding set of change primitives.

Module replacement also requires the execution of several change primitives, however, the execution of these change primitives will not result in a code inconsistency. Since it has been assumed that the modification has been designed correctly, the code in the new module must be consistent. Therefore, loading a new module cannot introduce an inconsistency. The deletion of a module cannot result in a code inconsistency either; there is no code to execute. And, since the reconfiguration of communication links must be implemented by overwriting data variables (i.e., no change to program code), the implementation of module update operations by module replacement maintains code consistency.

Directly overwriting a module can introduce other complications besides code inconsistencies. If the new version of the module is a different size than the original version, interesting memory management issues arise, especially if the new version is larger than the original. Thus, if updates to a module's code are implemented by overwriting data variables, the change management system must not only generate execution control instructions, it must also generate memory management instructions.

Updates to a module's data state cannot cause a code inconsistency, therefore, as far as data consistency is concerned, the choice of implementation technique (module overwrite or module replacement) is arbitrary. Various arbitration schemes could be used (e.g., one which results in the fewest number of change primitives), however, the actual arbitration scheme has no impact on the remainder of the change management process. In addition, unless variables are added or deleted as part of the update, the size of the original and new versions of the module will remain constant, and therefore it will be assumed that

all data updates are implemented via overwriting variables. Thus, the output of this part of the process is an unordered set of change primitives and, in the case of execution control, a set of control conditions as depicted in figure 5.2.

Figure 5.2 Management of Code Inconsistencies

5.5 Consistency of Message Transmissions

The correct transmission of messages requires the satisfaction of two conditions:

- a) The recipient of a message must exist. A situation where the recipient of a potential message transmission does not exist is defined as an *inconsistent configuration of communication links*; and
- b) The format and content of a message must be recognized by the recipient module. An *inconsistency in the interface between modules* exists if a module transmits a message whose format and/or content is not recognized by the recipient module (i.e., the message is not defined in the recipient module's input alphabet). Such a message will be misinterpreted by the recipient module.

5.5.1 Consistency of Communication Links

The consistency of communication links is dependent upon the order in which change primitives are executed. If a module is to be deleted, either as a stand alone operation or as part of a module update operation, it must first be effectively deleted from the program through the reconfiguration of communication links. If not, the possibility exists that one of its communications partners will transmit a message addressed to a non-existent module. The addition of a module also requires a reconfiguration of communication links, however, in this case, the reconfiguration of communication links must be performed after the addition of the new module. Thus, in order to maintain the consistency of communication links, the reconfiguration of communication links must be executed before the deletion of modules but after the addition of new modules.

For the purposes of maintaining the consistency of communication links, each modification operation is an independent action and therefore a total ordering of all change primitives is not required, only a partial ordering. For example, consider a modification involving the replacement of two modules, modules A and B. If intermodule communications are via a communications handler module¹² (message or procedure handler), each replacement operation requires the execution of three change primitives (load new, update data state of communication handler, delete old), for a total of six. These six change primitives may be executed in any order provided that the relative order of the change primitives associated with each replacement operation is maintained. Thus, if the change primitives are numbered as follows:

¹² The assumption of indirect intermodule communications is required only to limit the number of change primitives which must be executed to implement a reconfiguration of communication links.

- 1) load new version of module A;
- 2) update communications handler to redirect messages from old A to new A;
- 3) delete old version of A;
- 4) load new version of module B;
- 5) update communications handler to redirect messages from old B to new B; and
- 6) delete old version of B;

the only invalid sequences for the execution of the change primitives are ones in which:

- 2 is executed before 1;
- 3 is executed before 1 or 2;
- 5 is executed before 4; or
- 6 is executed before 3 or 4.

Thus, a valid sequence of execution primitives will maintain the consistency of communication links.

Ordering the sequence of execution of change primitives is not the only method of managing communication link consistency. For example, the deletion of a module requires the execution of at least two change primitives; the module removal primitive and one or more variable overwrite primitives to reconfigure communication links. Executing the module removal primitive before the variable overwrite primitive(s) will result in an inconsistent configuration of communication links. In this case the change management system must ensure that the communication partners of the deleted module will not attempt to transmit a message to it. In other words, the communications partners of the module to be removed must be prevented from transmitting messages (i.e., execution control) or the message transmission must be intercepted and redirected.

Intercepting and redirecting message transmissions is an alternative method of maintaining the consistency of communication links. The interception of message transmissions requires a reconfiguration of communication links and this reconfiguration of communication links will eliminate the possibility of a message transmission to a non-existent module. This reconfiguration of communication links is a new modification operation and once it is generated by the change management system it must be then fed back to the start of the change management process.

Figure 5.3 Management of Communication Link Inconsistencies

Execution control requires that the change management system generate a set of control conditions and actions which will prevent a module from transmitting a message. For example, if the communications partner is a task module and the operating system uses a round robin scheduling mechanism, the communications partner could be removed from the ready-to-run queue and be placed in a holding queue until communication links are reconfigured. This method totally restricts the communications partner from executing, when in fact the module only needs to be preventing from executing those portions of its code which would result in the transmission of a message to the module which had been removed. Regardless of the method chosen to prevent message transmissions, a set of control conditions/actions must be generated. The major drawback of execution control is

that all of the logical communications partners must be prevented from transmitting messages. If there are a large number of communications partners, this technique can severely impact system performance.

The options for managing the consistency of communication links are depicted in figure 5.3.

5.5.2 Module Interface Consistency

An inconsistency in the interface between a module and its communications partner(s) results from an update to the format or content of messages transmitted by or to the module. This type of modification to a program requires the implementation of two or more dependent modification operations. For example, consider a module which sorts (in ascending order) a list of objects. If this module is to be updated to provide the option of sorting in ascending or descending order (i.e., a change to the format of messages recognized by the module), not only must the sort module be updated, but all of its logical communication partners must be updated as well. These modification operations are dependent, and, regardless of the order in which the modification operations are performed, an inconsistency in the interface between modules will exist after the implementation of the first modification operation.

The most straightforward method of dealing with this type of inconsistency is to prevent the transmission of messages between the modules until all modules are updated; i.e., execution control. Again this method can severely affect system performance if the module has several communications partners. However, the inconsistency can be eliminated by temporarily adding an intermediate translation module to the program. The translation module, which must be added before any communication links are reconfigured, transforms the messages from the original to the new format or vice versa. The complexity of the translation module is dependent upon the change to the message and the order in which the modification operations are to be implemented. In the above example, a relatively simple translation module is required if the update to the sort module is performed first, whereas if the sort module is updated last, a relatively complex translation module is required. A modification implementation in which the sort module is updated in the middle of updates to the communications partners would require the most complex translation module. Sample pseudo-code for the translation modules is listed in Appendix B. Although a translation module can be designed irrespective of the order of

modification operations, it is assumed that a sequence of modification operations which requires the least complex translation module will be used.

Figure 5.4 Translation Module

With this assumption and the above example, the addition of the translation module would occur at the same time¹³ as the new version of the sort module. Communication links are reconfigured such that messages intended for the sort routine are directed to the translation module. The translation module reformats the message, by adding a default sort option (in this case, ascending) and forwards the revised message to the new version of the sort module. During the update of the communications partners of the sort module, communication links are updated such that message that had been redirected to the translation module are routed directly to the new version of the sort module. Once all the communication partners are updated, the translation module could be deleted. Figure 5.4 illustrates this sequence of events.

Figure 5.5 Managing Interface Consistency

¹³ The translation module may be added before or after the new version of the sort module but before communication links are reconfigured.

Updating the recipient module first (as in the above example) does not always result in the least complex translation module. Consider the same program after the modification and a second modification to reverse the effect of the previous modification; i.e., elimination of the ascending/descending option from the sort module. In this case, a sequence in which the sort module is updated after all its communications partners requires the least complex translation module.

Thus, the change management system must analyze the ordered set of change primitives for inconsistencies in the interface between modules. If inconsistencies are found, two options exist: (1) a set of control conditions can be generated; or (2) addition modification operations can be defined (the addition of a translation module, a reconfiguration of communication links, the deletion of the translation module, and a second reconfiguration of communication links). In the latter case, the additional modification operations must be fed back to the beginning to the change management process and a new ordered set of change primitives generated. This step in the change management process (depicted in figure 5.5) partially validates the ordered set of change primitives.

5.6 Data Consistency

Managing the consistency of data variables is the most complex function of the change management process. In a static modification scheme, data consistency is only an issue with respect to persistent data modules since all other data variables are in their initial state (which is consistent). The only option for managing the consistency of persistent data modules in a static modification scheme is to ensure that the modification design results in a consistent data state. For example, consider a simple airline reservation system (a logical model is depicted in figure 5.6 and pseudo code for the relevant modules is provided in figure 5.7) which assigns seats for each flight in ascending numerical order. The abstract modules, "date", each contain a persistent data variable since the correct execution of the program requires the value of the last seat assignment on a given flight to be maintained between invocations of the program.

If a modification is required to the seat assignment algorithm such that seats are to be assigned in descending order, the line $seat = seat + 1$ in the seat assignment module must be changed to $seat = seat - 1$ and the two constants, LastSeat and InitialSeat, must be assigned new values. This can be accomplished by replacing the scheduler module.

However, regardless of whether this modification design is implemented dynamically or statically, the modified version of the program will execute correctly only if the modification is implemented at a point in time when no data modules exist; i.e., none had yet been created or any that had been created were subsequently deleted by another module of the program. If it is implemented at any other point in time, the modified version of the program will execute incorrectly (eventually the same seat(s) will be assigned twice, thus causing the program to violate its functional specification) since the values of the seat variables in the existing persistent data modules are inconsistent with the new seat assignment module. Thus, in this example, the implementation of the modification must be delayed or the modification design must be reworked.

Figure 5.6 Airline Reservation System

The airline reservation example demonstrates the data consistency problem associated with persistent data variables in a static modification scheme. Since persistent data variables may not be in their initial state when change primitives are executed, a data inconsistency may exist. In a dynamic modification scheme the same problem exists, but on a larger scale; i.e., it is not limited to persistent data variables. Except for new modules added to the program, which are in their initial state, the values of data variables in other modules are unknown. Therefore, even in the case where a new module is added to the

program, the values of its data variables may be inconsistent with the data state of program in general.

Module Scheduler		; schedules flights
data variables	sender	; communications partner
	date	; date flight is to be scheduled
	new = constant	; initialization value for abstract modules
Begin Loop		
	recv-any(sender, date)	; wait for request to schedule flight
	create abstract module = date	; only one flight per day (this is a simple reservation system)
	write to date (new)	; initialize variable
	send to sender (ACK)	; optional depending upon the logical communications paradigm
End Loop		
Module Seat Assignment		; assigns seats
data variables	sender	; communications partner
	seat	; seat identifier
	new = constant	; indicator that no seats have been assigned
	LastSeatNo = constant	; indicator that all seats have been assigned
	FullFlight = constant	; error message
	InitialSeatNo = constant	; initial seat no to be assigned
Begin Loop		
	recv-any (sender, date)	; wait for request
	read date (seat)	; get value of last seat assigned
	if seat = new	; new flight?
	seat = InitialSeatNo -1	; initialize
	if seat = LastSeatNo	; is flight full?
	send-to-sender("assign",FullFlight)	; inform requester
	else	
	seat = seat + 1;	; get next available seat
	write date (seat)	; save seat assignment
	send-to-sender ("assign", seat)	; inform requester
End Loop		

Figure 5.7 Airline Reservation Code

5.6.1 Inconsistent Data State

A program's data state (i.e., the values of all data variables) provides at least a partial history of the program's execution (in the case of a program with persistent data modules, the execution history extends beyond a single invocation of the program). However, a data state does not provide a complete history of program execution; several different execution histories could result in the same set of values for the program's data variables. Nonetheless, the number of possible execution histories of a program is limited by the values of data variables.

An unmodified program is assumed to be executing correctly and therefore an inconsistent data state can result only from the execution of a change primitive. The execution of a change primitive can: create new data variables (load module primitive); eliminate existing data variables (remove module primitive); or update the value of a variable (variable overwrite primitive). An *inconsistent data state* exists when a data variable has a value (or no value in the case of a deleted variable) incompatible with all of the potential execution histories of the program given the value of all other data variables. Thus, in order to have a *consistent data state*, a variable affected by the execution of a change primitive must have a value which is compatible with at least one of the possible execution histories of the program.

5.6.2 Management of Data Consistency

Based on the definition of a consistent data state, it would appear that the change management system would have to verify the consistency of all data variables after the execution of every change primitives, however this is not the case. An inconsistent data state is not a sufficient condition to cause the program to execute incorrectly; the data variable with the inconsistent value must cause a module to alter one of the messages it transmits to one of its communications partners in order for the inconsistency to manifest itself. Therefore, the data state of a module affected by the execution of a change primitive only has to be consistent with the data state of its logical communications partners. In addition, if the communications between two modules does not depend upon the value of the variable which has been changed by the execution of the change primitive, the consistency of their respective data states is irrelevant.

Thus, if the execution of a change primitive would result in an inconsistent data state the change management system must first determine the relevance of the inconsistency. If it is relevant, there are three options for managing the inconsistency: (1) delay the execution of the change primitive; (2) update the inconsistent variable to a consistent value; or (3) isolate the modules which are inconsistent.

5.6.2.1 Delaying Execution

The execution of change primitives can be delayed until a consistent data state is reached. At this stage of the change management process, there is only an ordered set of change primitives, i.e., each change primitive has a relative time but not an absolute time

to be executed. In order for the design of the modification to be correct, there must exist some state of the program for which the data state is consistent (in the airline reservation example, this state arises when no date modules exist). Therefore, the change management system could monitor the state of the program and when the program enters a consistent data state, execute the change primitive. The change management system must also ensure that once a consistent data state is reached, the state remains consistent until the change primitive is executed.

For example, consider the unique id generation problem [1]. In this problem, a module generates ids and its functional specification requires that the last X number of ids generated be unique. This module could be implemented by issuing increasing integers, provided that the range of integers, R , is greater than X . Bloom examines two modifications to this module: one in which the module is replaced by an exact copy of itself, and one in which the replacement module issues ids in descending order. In the first case, the execution of the change primitive which reconfigures communication links must be delayed until the value the data variable containing the last id generated in the original version of the module is greater than or equal to X . If not, the initial value of this variable in the new version of the module will be inconsistent with the data state of modules which had requested ids from old version.

The major drawback to this approach is that a consistent data state may not be reachable from the current state of the program. In the second modification of the unique id generation problem, the execution of the change primitive must be delayed until the value the last id generated by the original version is between $0 + X$ and $R - X$, assuming that the initial value of the variable in the new version is R . If $X > R/2$, this state cannot be reached.

5.6.2.2 Updating to A Consistent Value

Rather than delaying the execution of a change primitive until a consistent state is reached, an alternative is to update the value of the inconsistent data variable. In the first modification of the unique id generation module, a consistent data state can be achieved by updating the initial value of the last id generated in the new module to the value of the same variable in the original module; i.e., by copying the data state of the original version to the new version. In the second modification, a simple copy will not suffice. One solution, assuming that $X \leq R/2$, is to update the value of the variable in the new version

such that it is equal to $x + X + 1$, where x equals the value of the last id generated by the original version.

This technique (as in the case of maintaining of module interface consistency) requires the implementation of additional modification operations (i.e., overwriting data variables). These additional modification operations must be fed back to the start of the change management process. In addition, the change management system must ensure that while it is implementing the update operation on a module, the data states of that module's communications partners do not change such that the module's updated data state is now inconsistent; i.e., control conditions must be generated. For example, in the unique id generation problem, the change management system must ensure that another module does not request an id before the new version of the module is updated and communication links are reconfigured.

5.6.2.3 Isolation

An inconsistent data state is only relevant if the value of a data variable in a given module is inconsistent with the value of a data variable in one of that module's logical communications partners. If a third module, whose data state is consistent with both of the other modules, is inserted between the other two, the program data state will be consistent; i.e., the two modules with inconsistent data states can be isolated from one another. This is essentially the technique used to maintain the consistency of module interfaces. An inconsistency in the interface between modules is actually the result an inconsistency in the data states of the two communications partners. The translation module isolates the two communications partners.

Inserting a module between the two modules with inconsistent data states is not the only method of isolating data inconsistencies. Communication links can also be reconfigured such that the two modules with inconsistent data states do not communicate. For example, in the airline reservation system, the new version of the seat assignment module is inconsistent with the existing date modules. Communication links can be reconfigured such that the new seat assignment module only communicates with date modules which are created after the reconfiguration of communication links. This requires a communications control module. The communication control module, upon receipt of a seat assignment request, checks to see if any seats have been assigned to that flight by the original version of the seat assignment module. If so, the request is forwarded to the

original module, otherwise it is forwarded to the new version. Once all date modules are consistent with the new seat assignment module (i.e., once all flights which had seat assignments performed by the original version of the seat assignment module have departed) communication links can be reconfigured such that all seat assignment requests are transmitted directly to the new seat assignment module. The original version of the seat assignment module and the communication control module can then be deleted.

5.6.3 Managing Data Inconsistencies

The change management system must examine the set of change primitives to determine if data inconsistencies exist. If found, three options are available to deal with the inconsistency. Depending upon the option chosen, monitor conditions, control conditions, and additional modification operations may have to be generated. The additional modification operation have to be fed back to the start of the change management process. The output of this final stage of the change management process, depicted in figure 5.8, is a valid set of change primitives, a set of control conditions, and a set of monitor conditions.

Figure 5.8 Management of Data Inconsistencies

5.7 The Change Management Process - Summary

The input to the change management process is a set of modification operations. The change management process involves generating a set of change primitives, establishing an order for the execution of these primitives, and validating the ordered set. In validating the order of change primitives, the change management system may also generate a set of control and monitor conditions. The number and complexity of these conditions is dependent upon the options selected for managing the various types of inconsistencies. The change management process is not linear; it includes a feedback loop from the validation stage of the process. Figure 5.9 summarizes the main functions of the change management process.

Figure 5.9 Main Functions of Change Management Process

Chapter 6

Model Validation

6.1 Introduction

Chapter 2 briefly described the work of previous researchers using their terminology and concepts. This chapter re-examines the previous research but uses the concepts and terminology of the dynamic change management process meta-model. This exercise demonstrates the validity of the meta-model and its sub-models.

There are many differences between the implementations of previous researchers, for example, whether the technique is applicable to procedure-based or message passing systems, whether the change management system is implemented as a separate process or as part of the operating system, and the size of the sub-unit of the program which is subject to change, *the unit of change*. However, these differences are ones of implementation, not principle. A dynamic change management system must update an executing program's code and data variables and ensure that the program executes correctly during the update.

The solutions examined in this chapter can be broadly classified into two groups; those which support all types of changes and those which only support the replacement of modules. The latter group is further subdivided predicated on whether the solution is operating system or language based. Operating system based module replacement solutions are examined first, followed by language based module replacement systems, and lastly complete change management systems.

6.2 Module Replacement

The majority of solutions proposed by previous researchers do not address all aspects of the change management process; most only offer a means of implementing a single modification operation, namely module replacement. Module replacement solutions can be further sub-divided into those which are implemented as a part of the programming language and those which are implemented as part of the operating system.

6.2.1 Operating System Solutions

This section examines operating system based solutions to module replacement. Implementing module update operations via module replacement maintains code consistency. These solutions also maintain the consistency of communication links by ordering the three change primitives (all solutions use some form of indirect addressing for procedures calls, thus the reconfiguration of communication links only requires the execution of a single data overwrite primitive) associated with this modification operation. In addition to only addressing the implementation of a single modification operation, the majority of previous researchers have simplified the change management process by restricting the types of changes which are supported. For example, in this group, only Segal & Frieder [11, 21, 22, 23] permit changes to the interface of a module; all other solutions maintain the consistency of module interfaces by not supporting these types of changes. Thus, the only real difference between these solutions is the manner in which potential data inconsistencies are managed.

6.2.1.1 DMERT

The solution offered by the field administrative subsystem of the DMERT operating system [25] does not address the issue of data consistency. The unit of change for this solution is a procedure (C language functions). The replacement procedure is loaded into a special scratch pad area; the *transfer vector*, which contains the address of the procedure is overwritten, and the original version of the procedure is deleted. The program, before and after the change, is illustrated in figure 6.1. The authors of this solution indicated that several challenges had to be overcome in order to implement the solution but they did not identify how these challenges were met.

Figure 6.1 DMERT

6.2.1.2 Dynamic Module Replacement

Dynamic Module Replacement [1] provides a limited capability to manage data inconsistencies during the replacement of a group of one or more guardians. The solution assumes that the initial data state of the replacement module (the program model permits a group of modules to be represented as a single module) is not consistent with the rest of the program. The data state is updated by copying the value of data variables from the original version of the module. This approach does not guarantee data consistency (as illustrated by the unique id generation problem), however, the author concludes that if the copy operation does not result in a consistent data state then the modification cannot be implemented dynamically.

Notwithstanding the limitation of simply copying data variables, in order to implement this solution, the change management system must ensure that the value of data variables do not change during the copy (update) operation. This solution uses execution control. The modules to be updated are halted and any ongoing transactions are aborted. The user then issues a series of commands to load the new modules, copy state information, update handles (reconfigure communication links), and restart the modules. Data state information is not copied directly to the new version. The Argus environment provides a mechanism for a guardian which has "crashed" to recover its state information from non-volatile memory. Thus, the values of data variables are copied to the non-volatile memory. The user, before issuing the restart command, must issue a command which informs the operating system that the module has "crashed". The operating system will recover the module's state information, thus completing the copy operation.

6.2.1.3 Data Restructuring

Two solutions, Changing Modules on the Fly [8] and the Dynamic Alterable System [12] use a conversion routine, which restructures abstract data types, designed into the replacement module to manage data consistency. In both cases, the program must be designed for dynamic change and thus system performance is affected even under normal operation.

6.2.1.3.1 Changing Modules on the Fly

In addition to a conversion routine in the replacement module, this solution uses two levels of address indirection, a data locking mechanism, and embedded version numbers. Figure 6.2 depicts a logical communications model of this approach. Under normal operation, the calling module sends a message to the indirect module, which locks the instance of the data type to be operated on, and forwards the message to the code segment module. The code segment verifies the version number of the data instance and begins execution.

Figure 6.2 Changing Modules on the Fly

In order to replace a module, new versions of the indirect and code segment modules and a conversion routine module¹⁴ are loaded into memory. The capability address is overwritten to point to the new version in a similar manner to the overwriting of

¹⁴ Alternatively, the conversion routine can be modeled as a part of the code segment; i.e., the code segment and conversion routine are combined and modeled as a single module as in figure 6.2.

transfer vectors (figure 6.1). Thus, the next time that the module is accessed, the version number in the data instance will not correspond to the version number expected by the new code module; i.e., an incorrect version number indicates the presence of an inconsistent data state. The code module calls the conversion routine to update the data structure to consistent state.

With this approach, it is impossible to ascertain when a modification is complete (the implementation of the modification is not complete until all instances of a data structure have been updated and there is no way to determine this in this solution) and therefore the conversion routine can never be deleted even if it is no longer required. In addition, subsequent modifications would require both old and new conversion routines since a particular instance of a data structure may not have been accessed by the first replacement module before it is accessed by the newer replacement module. Thus, the size of the program will grow uncontrollably.

This solution must also account for the possibility, although remote, that an instance of a data structure may have been updated to an even newer version. This case is handled by the second level of indirection. If the code segment finds that the version number of the data instance is higher than expected, a message is sent to the indirect segment. This message would be directed to the newer version of the indirect module since all message are physically transmitted via the capability address module and this module would have been updated.

6.2.1.3.2 Dynamic Alterable System

The Dynamic Alterable System (DAS) provides three module update operations. The *chlink* and *repcode* operations do not address data consistency; i.e., the user must determine that updating a module's code will not result in a data inconsistency before performing these operations. The two separate operations are required because of the DAS addressing scheme. A DAS module consists of code, object and own data segments (each segment is represented as a separate module in the program model). If a DAS module does not have any data segments a new descriptor is loaded into memory and the contents of the two descriptors are exchanged by the *repcode* operation (figure 6.3). The *chlink* operation (figure 6.4) overwrites the link in the descriptor which points to the code segment.

Data consistency is addressed by the *repmat* operation (figure 6.5). The new code and data segments (and descriptors) are loaded into memory. A special data restructure procedure is executed which copies, and restructures if necessary, the data from the old to the new version of the data segment modules. The data restructure procedure requires special in and out routines in the new and old versions of the module, thus, the original version of the program must have been designed for dynamic change. Once the data is copied, the descriptor links are overwritten to point to the new code segment and the original code and data segments can be deleted. Execution control is used to prevent execution during the operation and is implemented by setting a special flag in the descriptor.

Figure 6.3 Repcode Operation

Figure 6.4 Chlink Operation

Figure 6.5 Repdat Operation

6.2.1.4 PODUS

The Procedure-Oriented Dynamic Updating System (PODUS) proposed by Segal & Frieder [11, 21, 22, 23, 24] permits the replacement of procedures by overwriting addresses in a procedure *binding table*. Data consistency is handled by translation modules

and by updating the data state. Translation modules, termed *interprocedures* by Segal and Frieder, are used (as described in section 5.3.2.2) to handle updates which involve a change to the interface to a module.

A similar concept is used to handle data state inconsistencies. Mapper procedures copy and update the value of static variables from the old to the new version of a module. To ensure that the value of data variables cannot change during the copy/update operation, all communication link reconfigurations are incorporated (i.e., the binding table overwrites) when the replacement module is in a receive-any state. This is implemented by halting the program after procedure return instructions. The change management system examines the runtime stack to determine which procedures may be updated. Only those procedures which are not on the stack (i.e., those in a receive-any state) can be updated. In addition, a procedure cannot be updated if it can call any other procedure which is on the stack. The latter restriction prevents a new procedure from calling an old procedure and thus ensures that low level procedures are updated before higher level procedures. Therefore, only one set of translation modules is required. The restriction thus orders the implementation of modification operations.

This solution is more elegant than the other module replacement solutions in that it can handle a wider range of data inconsistencies, it does not require the program to be designed for change, and it does not affect system performance when a modification is not being implemented. However, the solution is not complete. The main procedure of a program can never be updated since the solution requires a procedure to enter a receive-any state (which the main procedure of a program never does) before it can be replaced. This restriction also unnecessarily delays the replacement of procedures with non-static data variables. A procedure which does not contain static data variables nor access any global variables can be updated at any time. Consider the case where the communication links to such a procedure are reconfigured when it is in the middle of executing. The execution of the original module will not be affected by this change. All future calls to the procedure will be directed to the new version. The replacement module will begin execution in a receive-any state. Its correct execution does not depend upon the value of local data variables; if the future states of a procedure model in a receive-any state depended upon the current value of a data variable, that variable would be, by definition, a static variable.

6.2.2 Language Based Solutions

Several previous researchers have proposed solutions which rely upon the special properties of a particular language or class of language to replace modules. In general these solutions require that the program to be updated be designed for change since the change management system, or portions of it, is designed into the program. In addition, the management of data consistency is limited to copying the data state from the original to the replacement module. By concentrating on the special properties of a particular language or class of languages, these solutions overlook the fact that once a program has been compiled and is executing, the original language in which it was written is immaterial to the change management process.

6.2.2.1 Object Oriented Languages

The object oriented solution proposed by Stadel [25] requires that the program to be updated contain a configuration manager from which a user can enter maintenance commands. All modules must be an instance of a deferred class and have a cloning procedure. Cloning may be done in the foreground or in the background. If done in the foreground, all other execution is suspended; i.e., execution control is exercised. Cloning involves copying the state of the original module to the new version, thus, the change must be implemented when the values of data variables in the original module are consistent with the new version. Stadel does not indicate how this is determined.

Background cloning is performed by a separate module which is loaded at the same time as the new version. Communication links are reconfigured such that messages are directed to the cloning module. The cloning module retransmits the message to both the old and the new versions of the module. The transmission of the messages to both versions of the module has the effect of copying variables provided that the replacement module is a container class. Once all the data variables have been copied to the new version, the two versions are kept synchronized until communication links are reconfigured to transmit message directly to the new version. Thus, this approach is an example of the isolation technique described in section 5.4.2.3.

The loading and deleting of modules and the reconfiguration of communication links is performed by a dynamic link loader. The dynamic link loader is provided by the operating system and is called by the configuration manager.

6.2.2.2 Secure on the Fly Modifications

This solution is quite similar to the previous one. A reconfiguration manger module performs the functions of both the configuration manager and the dynamic link loader. The main differences are that the conversion routine is termed a *delegation routine* and this routine is part of the original module rather than the new version. An update is implemented by a call to the delegation routine of the original module. The delegation routine *syphons* the data state of the old module into the new module. When syphoning is complete, communication links are reconfigured and the original module is deleted.

6.2.2.3 Ada

Burns & Wellington [3] have suggested two options for transferring state information during the replacement of passive Ada modules. In both cases, execution control is used to prevent the module being replaced from executing during the update. The first option is to split the module in two: a code module and a data module and just replace the code module. The second option is to copy the data into a third module, a disk file, and the copy it again into the new version of the module. The replacement of active Ada modules was not considered. The authors concluded that the transfer of state information during the replacement of active modules cannot be described in general terms; each replacement must be treated as a special case.

6.3 Change Management Systems

The two solutions examined in this section are complete change management systems; i.e., they address the addition and deletion of modules in addition to module replacement. The Dynamic Modification System (DYMOS) [6, 18] is a procedure based solution whereas Kramer & Magee's configuration manager [14, 15, 16, 19] is applicable to a message passing system.

6.3.1 Dynamic Modification System

DYMOS requires that the user perform part of the change management process. The user enters a series of commands to add, delete, and update modules (procedures or STARMOD modules which consist of procedures and data structures). The user must ensure that the implementation of each command will leave the program in a consistent state. Module replacement is used to maintain code consistency, however, the

management of other types of consistencies is the responsibility of the user. For example, if a module is to be deleted, the user must also specify the modules to be updated to maintain the consistency of communication links.

The user has the option of using execution control or translation modules to manage the consistency of module interfaces. Data inconsistencies are managed by a conversion routine designed into the replacement module. Execution control ensures that the data state of the original module does not change during the copy/update process.

Execution control is implemented by a lock bit and use count stored in the first memory location of every module. Every procedure call increments and decrements the use count on entry and exit to a procedure. If it is necessary to prevent the execution of a module during the implementation of a command, the user specifies the module in the command. Consider the updating of a sort routine as described in section 5.3.2.2 and assume that this module is called by two other modules, A and B. This update could be implemented by the command "update sort, A, B when A, B idle."

The implementation of this command is depicted in figure 6.6. Upon receipt of the command, the change management system would monitor the use count of modules A and B. When the use count reached zero, the lock bit would be set preventing the execution of the module. The new sort module is loaded into memory and communication links to the modules are reconfigured. DYMOS uses indirect addressing and thus communication links can be reconfigured by overwriting the value of variables stored in Module Address Tables and Procedure Address Tables. Once communication links are reconfigured the original modules can be deleted.

Alternatively, the consistency of communication links could be maintained by means of a translation module. The translation module in this example would add a default sort order. In this case, the address table entry for the original module is overwritten to point to the translation module and a new entry is added for the replacement module. The calling modules are updated to use the new module. Once all the callers are updated, the translation module and the original version of the sort module may be deleted. The changes to the address table are depicted in figure 6.7.

Step one:
new modules
loaded and
lock bits
set

Step two:
communication
links are
reconfigured

Step three:
original
modules are
deleted

Figure 6.6 DYMOS Execution Control

Figure 6.7 Module Interface Consistency

Conversion routines are used to maintain data consistency. The conversion routine copies and updates if necessary, the value of data variables from the original to the new module. Execution control is used to ensure that the values of data variables do not change during the conversion process. This five step update operation is depicted in figure 6.8: (1) the replacement and conversion modules are loaded; (2) the lock bit in the original version is set to prevent further execution; (3) the convert module begins execution to copy/update data variables; (4) communication links are reconfigured by overwriting the address table; and (5) the original module is deleted.

Figure 6.8 Managing Data Consistency

Use counts and lock bits are necessary to implement execution control, however, their use affects system performance under normal conditions. Before every message transmission, the lock bit must be checked and the use count incremented or decremented. In addition, the correct execution of the program cannot be guaranteed since an operator error could permit the introduction of inconsistencies.

6.3.2 Kramer & Magee

The configuration management system proposed by Kramer & Magee [14, 15, 16, 19] eliminates the possibility of operator error during the implementation of a modification. This system requires that the program be described in a specification language which describes the logical and physical structure of the program. Changes to the system are submitted in the form of change specifications. Change specifications identify the modules to be deleted, added or updated and the logical communication links which must be deleted or added. The system validates the change specifications and derives a set of management primitives which are implemented by the operating system. This process is outlined in figure 6.9.

Figure 6.9 Configuration Management System

Four management primitives are used: (1) create, which loads a new module; (2) delete, which removes a module; (3) link, which creates a logical communications link; and (4) unlink, which deletes a logical communication link. The create and delete primitives are executed by the operating system whereas the link and unlink primitives are implemented by sending a message to the module which transmits messages over the link in question. Execution control is used to maintain the consistency of communication links and is implemented by sending messages directing modules to a passive state. A module in a passive state may receive but not transmit messages. Thus, in order to implement a modification operation, the module directly affected by the operation (i.e., the module to be added, deleted or updated) and all of its communication partners are directed to the passive state. The module directly affected by the operation will enter a receive state and,

since none of its communication partners will transmit it a message, it will only respond to messages from the change management system. Once all communication links are reconfigured, modules are directed back to an active state.

This system requires extensive redesign of the program to support dynamic change. In their evolving philosophers example, which uses a variation of the hygienic solution [5] to the classical dining philosophers problem, the authors describe how philosophers can be added or deleted from the group while the program is running. The hygienic solution requires that each philosopher follow seven rules which dictate when it can request a fork, send a fork, or change from a thinking to hungry or eating state. In order to support the dynamic addition or deletion of philosophers, one rule must be changed and six new rules added.

Data consistency is also managed by program design. In the evolving philosophers example, data consistency is dependent upon the value of the fork and fork_request variables in each philosopher module. When a communications link is established between two philosophers (after the deletion or addition of a philosopher) the value of these variables in each philosopher module is initialized based upon a total ordering of all philosopher modules.

6.4 Summary

The previous solutions for dynamically modifying a computer program can be explained by the theory of dynamic change. Modification operations have been implemented in a similar manner by all previous researchers. All of the solutions manage code and communication link inconsistencies in the same manner, namely by using module replacement and ordering the execution of change primitives. Interface consistency is only addressed by two of the solutions; all of the rest do not support changes to the interface of a module. Thus, the only essential differences in the implementations have been the approach to the management of data inconsistencies. Furthermore, none of the solutions have completely addressed data inconsistencies; some have totally ignored it while others have only addressed special cases such as copying the values of variables from the original to the replacement module. The approaches to the management of data inconsistencies are summarized in table 6.1.

Solution	Management of Data Inconsistencies
DMERT	ignored
Dynamic Module Replacement	change management system copies data variables from original to the replacement module
Fabry	conversion routine in replacement module
DAS	conversion routine
PODUS	interprocedures which copy and update data variable from original to replacement module
OO Language	change management system copies data variables from original to the replacement module
Secure on the Fly Modifications	replacement module is passed as a parameter to original module which essentially copies the values of variables
Ada	change management system copies data variables from original to the replacement module
DYMOS	interprocedures which copy and update data variable from original to replacement module
Kramer & Magee	ignored

Table 6.1 Management of Data Inconsistencies

Chapter 7

Conclusions

7.1 Theory of Dynamic Change

Software systems must be continually updated. The updating of a program is not limited to incorporating changes to the source code; the current executing version of the program must also be replaced by the new version. One method of minimizing the disruption associated with this replacement is to implement the modification dynamically. Previous research into dynamic change has concentrated on implementation and neglected theory. This dissertation has examined the theory of dynamic change.

The development of the theory of dynamic change has resulted in three key findings: (1) a generic model of an executing program; (2) the identification of change primitives; and (3) a definition of correctly executing program applicable to a program undergoing modification.

7.1.1 Model

An executing computer program is modeled as a collection of communicating extended state machines. A complete representation of a program requires the inclusion of portions of the operating system and the hardware devices that the program directly accesses. Each state machine (module) corresponds to a portion of code and data, a memory location, or a hardware device.

The modules communicate via a message passing paradigm based upon three communication primitives: (1) a non-blocking send; (2) a selective blocking receive; and,

(3) a non-selective blocking receive. These primitives describe the physical flow of information between modules. Intermodule communications can also be viewed at a higher, logical level. The simplest logical flow, a jump, can be modeled as the execution of a single send by the originator and a receive by the recipient module. More complex information and control flows require the execution of multiple sends and receives and will normally include intermediate communication handler modules. Communication handler modules are operating system modules.

7.1.2 Change Primitives

The dynamic implementation of change has two requirements: to update the program to the new version; and, to ensure the correct execution of the program during the implementation. A modification to a computer program is implemented as a series of smaller changes (modification operations) to the modules which make up the program. This is the same technique used in a static modification scheme. A modification operation may be incorporated by replacing the module or by overwriting some or all of the memory locations that it occupies. The replacement of a module requires the execution of two primitives; loading the new version and deleting the original. The third primitive is the overwriting of a single memory location. The executions of a series of these three primitives are the only actions that must be performed to implement a modification. All other activities during the dynamic modification of a program are to ensure the correct execution of the program during the update; i.e., the management of change.

The operand of a change primitive is another module or a component of another module and therefore, the execution of change primitives is a meta-level operation. In designing a change management system, the meta-level module(s) which executes change primitives may be located in a separate process, included as components of the operating system, or may be incorporated in the program to be updated.

The main disadvantage of incorporating the change implementation modules in the program to be updated is the overhead. The code is always present even though it is only executed relatively infrequently and each program must have its own copy. Locating these modules in a separate process eliminates these disadvantages but, some operating systems may not permit the existence of meta-level models which cross process boundaries. In fact, some operating systems do not permit meta-level operations within a process. Therefore, the goal of designing a change management system which is independent of

programming language and operating system is not achievable. However, the development of a generic system which only requires minor tailoring for a particular environment should be possible.

7.1.3 Consistency

A modification to a computer program may involve a corresponding change to the functional specification that the program must satisfy. Therefore, the definition of correctly executing program as one which satisfies its functional specification is not applicable during the implementation of a modification. The new definition of a correctly executing program is based upon the absence of error conditions; i.e., inconsistencies. An inconsistency can exist only in the components of a module (the state transition function, the output transition function; or the module's data state) and inconsistencies can be introduced only by the execution of a change primitive.

In a correctly designed modification, state and output transition function inconsistencies are eliminated by the subsequent execution of another change primitive. The change management system can eliminate these inconsistencies through the selection and ordering of change primitive execution or, if the execution of a primitive does introduce an inconsistency, halt the execution of that module until the subsequent change primitive, which removes the inconsistency, is executed. Data inconsistencies are more difficult to manage; data inconsistencies may still exist even after the execution of all change primitives. The change management system may have to add new modification operations, such as the addition and subsequent deletion of data conversion modules, or it may have to delay the execution of a change primitive until a module's data state is consistent.

7.2 Validation

The work of previous researchers has been used to validate the theory of dynamic change. Although most of the previous work has concentrated on the implementation of a single modification operation, specifically the replacement of a module, their work has been explained in terms of the program model, the execution of change primitives, and the management of inconsistencies.

The program model is independent of operating system, programming language, and addressing scheme since the modules and communication primitives are based upon

the execution of binary code by the CPU. Several different operating systems, programming languages and addressing schemes were described by previous researchers and the model was used to explain each of them. Multiprocessor environments are the only significant arena not adequately addressed in the literature. The model should be applicable in this environment since it does not include any assumptions about the number of CPUs, however, this issue has not been examined extensively.

In using the model it was observed that modification operations have been implemented in a similar manner by all previous researchers. The only essential differences in their implementations have been the approach to the management of data inconsistencies. The management of data inconsistencies has not been completely addressed by any of the previous researchers; some have totally ignored it while others have only addressed special cases. Thus, this is the one area of the theory which has not been well validated. The definition of a consistent program data state is based on the premise that if the data state of each module is consistent with the data state of its logical communications partners then the data state of the program is consistent. This definition may not be applicable to a distributed program in which multiple instances of modules exist. The program model does permit, in fact requires, multiple instances of a module, however, each instance of a module executes the same code. If a program with distributed code is updated, the data states of each copy of a module may have to remain consistent, even though there is no logical communication link between the two modules.

7.3 Implementation

The implementation of a dynamic change management does not have to be completely automated; human decision making can be included in the design of the system. All previous solutions included human decision making. The level of human involvement varies widely. In the most highly automated solution, the configuration manager of Kramer and Magee [15], human decision making is only required during the design of modules. Human input is required to ensure data consistency since this solution assumes that the data state of any new or replacement module is consistent. Other solutions, such as DYMOS [18], require significant human input. DYMOS permits the operator to determine the order in which modification operations are implemented, the methodology for controlling consistency (the maintenance of consistency or execution control) and requires a programmer to design, code and test translation modules and conversion routines. Even those solutions which have simplified the change management

process by not supporting certain types of changes (e.g., module interface changes) require the examination of the modification to be implemented to ensure that it does not include an unsupported change. While this activity could be automated, none of the previous solutions have chosen to do so.

The objective of this dissertation was to identify the theory of dynamic change and identify the options for implementing various types of change to facilitate the engineering of dynamic change management systems. The change management process is a set of activities (e.g., generating the set of change primitives, ordering change primitives, validating data consistency, and the execution of change primitives) and it should be possible to completely automate all the activities. However, these activities, with the exception of the actual execution of change primitives, could also be partially automated or manually implemented. The level of direct or indirect human involvement in the change management system is an engineering design decision.

7.4 Future Work

Dynamic change can be segregated into two separate activities; the implementation of change and the management of change. Several different methods of implementing change (i.e., the execution of change primitives) have been devised although most are only applicable under a specific set of circumstances. Future development should concentrate on the management of change; the generation of a valid ordered set of change primitives and control conditions that could be passed to an implementation system which would automatically update the program. The configuration manager of Kramer & Magee would be a useful starting point. This system will generate a set of ordered change primitives, however, it does not address data consistency. The identification of control conditions and the generation of conversion routines are required to address data consistency.

Such a change management system would require a set of modification operations as its input. The model of a computer program requires that module boundaries be determined from an examination of object files not source files. An examination of source files is not sufficient since optimizing compilers can obscure module boundaries and the extent of changes to incorporate may dictate the replacement of an entire task as a single modification operation rather than the replacement of all of the task's individual procedures. The development of an automated tool which determines module boundaries

and the set of modification operations is also an area for future research. Such a tool would provide language independence.

Given a change management system which outputs a valid, ordered set of change primitives and control conditions, the design of an implementation system would be the next and last step. For efficiency reasons, the majority of this system should be located in a separate process, only the modules which execute change primitives should be designed into the operating system.

The design of a change management system as described above would achieve language independence and preclude the requirement to design support for dynamic change into the original version of the program to be updated. The design would require minor modifications (the addition of modules which execute change primitives) to the operating system.

References:

- [1] T. Bloom, "Dynamic Module Replacement in a Distributed System," MIT Laboratory of Computer Science, MIT/LCS/TR-303, March 1983.
- [2] R. Boute, J. De Man, H. Peeters, "Secure On the Fly Software Modifications," *Fourth International Conference on Software Engineering for Telecommunication Switching Systems*, July 1981, pp. 49-53.
- [3] A. Burns, A.J. Wellings, "Dynamic Change Management and Ada," *Journal of Software Maintenance*, Vol. 1, 1989, pp. 121-131.
- [4] P.M. Cashin, et al, "Experience with a Modular Typed Language: PROTEL," *Fifth International Conference on Software Engineering*, 1981, pp. 136-143.
- [5] K.M. Chandy, J. Misra "The Drinking Philosophers Problem," *ACM Transactions on Programming Languages*, Vol. 6, No. 4, October 1984, pp. 632-646.
- [6] R.P. Cook, I. Lee, "DYMOS : A Dynamic Modification System," *ACM SIGPLAN Notices*, Vol. 18, No. 8, August 1983, pp. 201-202.
- [7] F. DeMerer and H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, June 1976, pp. 80-86.
- [8] R. S. Fabry, "How to Design a System in which Modules can be Changed on the Fly," *Proceedings of Second International Conference on Software Engineering*, October 1976, pp. 470-476
- [9] R. S. Fabry, "Capability-Based Addressing," *Communications of the ACM*, Vol. 17, No. 7, July 1974, pp. 403-411.

- [10] D.G. Foxall, et al, "PROTEL: A High Level Language for Telephony," *Third International Computer Software and Applications Conference*, 1979, pp. 193-197.
- [11] O. Frieder et al, "Dynamic Program Modification in Telecommunications Systems," *Seventh International Conference on Software Engineering for Telecommunication Switching Systems*, July 1989, pp. 168-172.
- [12] H. Goullon, R. Isle, and K. Lohr, "Dynamic Restructuring in an Experimental Operating System," *IEEE Transactions on Software Engineering*, Vol. 4, No. 4, July 1978, pp. 298-306.
- [13] F.G. Hennie, *Finite-State Models for Logical Machines*, John Wiley and Sons, New York, 1968
- [14] J. Kramer, J. Magee, "Environment Requirements For System Evolution," *Proceedings of Software Process Workshop*, 1984, pp. 77-87.
- [15] J. Kramer, J. Magee, "The Evolving Philosophers Problem, Dynamic Change Management," *IEEE Transactions on Software Engineering*, Vol. 16, No. 11, November 1990, pp. 1293-1306.
- [16] J. Kramer, J. Magee, "Dynamic Configuration for Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. 11, No. 4, April 1985, pp. 424-436.
- [17] D.M. Lasker, "Module Structure in an Evolving Family of Real Time Systems," *Fourth International Conference on Software Engineering*, 1979, pp. 22-28.
- [18] I. Lee, *DYMOS: A Dynamic Modification System*, Ph.D. Dissertation, University of Wisconsin-Maddison, 1983.
- [19] J. Magee, J. Kramer, "Dynamic Configuration for Distributed Real-Time Systems," *IEEE Real-Time Systems Symposium*, 1983, pp. 277- 288.
- [20] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, New York: McGraw-Hill, 1982.
- [21] M. Segal, O. Frieder, "Dynamic Program Updating in a Distributed Computer System," *Proceedings of the IEEE Conference on Software Maintenance*, October 1988, pp. 168-172.

- [22] M. Segal, O. Frieder, "Dynamically Updating Distributed Software: Supporting Change in Uncertain and Mistrustful Environments," *Proceedings of the IEEE Conference on Software Maintenance*, October 1989, pp. 254-261.
- [23] M. Segal, O. Frieder, "Dynamic Program Updating: A Software Maintenance Technique for Minimizing Software Downtime," *Journal of Software Maintenance*, Vol. 1, 1989, pp. 59-79.
- [24] M. Segal, O. Frieder, "On-the-Fly Program Modification: Systems for Dynamic Updating," *IEEE Software*, Vol. 10, No. 2, March 1993, pp. 53-65.
- [25] Stadel, "Object Oriented Programming Techniques to Replace Software Components on the Fly in a Running Program," *ACM SIGPLAN Notices*, Vol. 26, No. 1, January 1991, pp. 99-108.
- [26] R.H. Yacobellis et al, "The 3B20D Processor & DMERT Operating System: Field Administration Subsystem," *The Bell System Technical Journal*, Vol. 62, No. 1, January 1983, pp. 323-340.

Appendix A

Updating the Main Procedure of a Program

One criticism applicable to the majority of previous change management systems is that they cannot be used to update the main procedure of a program without resorting to execution control. Module replacement requires the execution of a minimum of three change primitives: (1) loading a module; (2) removing a module; and (3) a variable overwrite to reconfigure communication links (more than one variable overwrite primitive may have to be executed if an intermediate communications handler module is not provided by the operating system). Generally, in the case of procedural programs, previous solutions implement the reconfiguration of communication links by overwriting a variable in a procedure handler module. Even if the address of the main procedure is stored in the procedure handler module, updating its value will not serve any useful purpose since the main procedure of a program is only called once; by the operating system when the program is started. The problem with this type of approach is that it assumes that the module to be replaced is a recipient of information exchanges, not the originator.

The link from the operating system is not the only incoming link to the main procedure of a module. When a procedure calls another procedure it enters a receive-specific state to await a reply message. The reply message is transmitted over a link from the called procedure to the calling procedure. The called procedure (recipient module) identifies the link to be used for the transmission of the reply message by examining the stack; i.e., the return address on the stack corresponds to the communications link to the originator module. Thus, the communication link can be reconfigured by overwriting the return address to point to the new version of the main procedure as illustrated in figure A.1.

Figure A.1 Updating Return Address

Such an approach would require careful timing of the execution of the memory overwrite primitive. In addition, this approach does not address data consistency. A more elegant approach involving the temporary addition of a reconfiguration module is depicted in figure A.2. The replacement of the main procedure would proceed as follows:

(1) the replacement main procedure and the reconfiguration module are loaded into memory;

(2) the communication link to any procedure, ideally a non-reentrant procedure which is called regularly, is reconfigured to the reconfiguration module;

(3) the reconfiguration module would overwrite the return address on the stack, copy and update any static variable from the original version of the main procedure to the new main, and call the procedure that would have been called before the reconfiguration of communication links;

(4) after the return from the call, the reconfiguration module would signal the configuration manager and send a reply message to the new version of main; and

(5) the configuration manager would delete the reconfiguration manager and the original version of the main procedure.

A resolution test chart featuring various patterns of vertical and horizontal lines. The patterns are arranged in columns, each associated with a numerical value representing resolution. The values include 1.0, 1.1, 1.25, 1.4, 1.6, 1.8, 2.0, 2.2, 2.5, 2.8, 3.2, 3.6, 4.0, 4.5, 5.0, 5.6, 6.3, 7.1, and 8.0. The lines become progressively thinner and closer together as the numerical value increases.

NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

Figure A.2 Replacing Main Procedure

Appendix B

Translation Modules

A program containing a sort module is illustrated in figure B.1. A modification to add the option of sorting in ascending or descending order requires an update to the sort module and both of the original modules, A and B. If module interface consistency is to be maintained, a translation module must be temporarily added to the program. The complexity of this module is dependent upon the order in which the modules are to be updated. There are three orders which must be considered: (1) the sort module is updated first; (2) the sort module is updated last; and (3) one of the originator modules is updated, then the sort module, and finally the other originator.

Figure B.1

In the first scenario, the translation module is relatively simple as the translation module only needs to add a default sort option. Pseudo-code for this scenario is listed below.

Module Translator		; Translates Messages
data variables	sender	; message originator
	objects	; list of objects to be sorted
	order = constant	; default sort order
Begin Loop		
	recv-any (sender, objects)	; wait for request
	send-to-sort (my_id, order, objects)	; add default order and forward message
	recv-sort (objects)	; wait for reply
	send-to-sender (objects)	; forward reply
End Loop		

In the second scenario, the translation module must examine the sort order requested by the originator module and reverse the sort if required.

Module Translator		; Translates Messages
data variables	sender	; message originator
	objects	; list of objects to be sorted
	order	; sort order
Begin Loop		
	recv-any (sender, order, objects)	; wait for request
	send-to-sort (my_id, objects)	; forward message
	recv-sort (objects)	; wait for reply
	if order = ascending	; no translation required
	send-to-sender (objects)	; forward reply
	else if order = descending	; translation required
	reverse (objects)	; reverse order of objects
	send-to-sender (objects)	; forward reply
	end if	
End Loop		

The third scenario results in the most complex translation module. The translation module must be able to handle four situations: (1) both the sort module and the originator have been updated; (2) neither the sort module nor the originator have been updated; (3) the sort module has been updated but not the originator; and (4) the originator has been updated but not the sort module.

In the first and second cases, the translation module forwards the message and the reply without modification. In the third case, a default sort order must be added and in the fourth case, the reply may have to be translated (reversed). The translation module must be aware of the update status of its communications partners before it can decide what type of translation, if any, is required.

The status of the sort module can be handled by including a variable in the translation module which is overwritten by the change management system when the sort module is updated. Although this technique could also be used to track the update status of originator modules, a separate variable is required for each originator and therefore, if there are several originator modules, this solution could become very cumbersome. An alternative is to determine the update status of the originator from the format of received messages. This approach may require that the translation module be implemented in a low level language, i.e., assembler.

Module Translator		; Translates Messages
data variables	sender	; message originator
	message	; variable format message
	order	; sort order
	sort = unmodified	; status of sort module
Begin Loop		
	recv-any (sender, message)	; wait for request
	if message_format = (order, objects)	; originator updated
	if sort = modified	; case 1
	send-to-sort (my_id, order, objects)	; forward message
	recv-sort (objects)	; wait for reply
	send-to-sender (objects)	; forward reply
	else if sort = unmodified	; case 4
	send-to-sort (my_id, objects)	; forward message
	recv-sort (objects)	; wait for reply
	if order = ascending	; no translation required
	send-to-sender (objects)	; forward reply
	else if order = descending	; translation required
	reverse (objects)	; translate reply
	send-to-sender (objects)	; forward reply
	end if	
	end if	
	else if message_format = (objects)	; originator not updated
	if sort = unmodified	; case 3
	send-to-sort (my_id, objects)	; forward message
	recv-sort (objects)	; wait for reply
	send-to-sender (objects)	; forward reply
	else if sort = modified	; case 2
	order = ascending	; add default order
	send-to-sort (my_id, order, objects)	; forward message
	recv-sort (objects)	; wait for reply
	send-to-sender (objects)	; forward reply
	end if	
	end if	
End Loop		

END

2007-94

FIN