

# ON-LINE SOFTWARE VERSION CHANGE

*A Thesis Submitted*

*in Partial Fulfillment of the Requirements*

*for the Degree of*

*Doctor of Philosophy*

*by*

*Deepak Gupta*

*to the*

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

*November 1994*

# CERTIFICATE

Certified that the work contained in the thesis entitled “*On-line Software Version Change*”, by “*Deepak Gupta*”, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

---

(Dr. Pankaj Jalote)  
Department of Computer Science &  
Engineering,  
Indian Institute of Technology,  
Kanpur.

---

(Prof. Gautam Barua)  
Department of Computer Science &  
Engineering,  
Indian Institute of Technology,  
Kanpur.

November 1994

# Synopsis

On-line software version change has emerged in the recent years as a technique for installing software changes in systems in which the shutdown necessitated by the conventional method of installing a new version of a running program can not be tolerated. In the present thesis, we present a formal framework for modelling changes to running programs and use it to study the *validity* of an on-line change. We also discuss some implementation issues and describe a prototype implementation.

Our framework for on-line changes is based on the notion of processes and process states. An on-line change is defined as a sequence of actions involving stopping a running process, changing its code, mapping its state and then continuing it. The overall behavior of a process that has undergone an on-line change is likely to be different from any behavior that can be exhibited by either of the two program versions. It is, therefore, important to define what constitutes an “acceptable” behavior of such a process. We capture this notion in our definition of the *validity* of an on-line change. We define an on-line change to be valid if some time after the change, the process reaches a reachable state of the new program version. Thus, validity ensures that following a change, the process starts behaving like the new version of the program after a “transition period”.

We first consider validity of on-line changes to programs written in sequential procedure based languages. For this purpose, a very simple model in which procedures and functions are not allowed is first considered. State is modelled as a mapping from variable names to values. For this model, we show that it is undecidable to find whether or not a given on-line change is valid. This result has important consequences. It means that computable necessary and sufficient conditions for validity of change can not be obtained. Undecidability in this simple model also

implies undecidability in other more realistic models. We then develop a set of sufficient conditions which ensure validity and show how they can be checked using data-flow analysis. We then extend the model to include functions and procedures and consider a function (or procedure) as the unit of change. For this model, we develop easily checkable sufficient conditions for ensuring validity that are based on the presence or absence of functions on the run-time stack.

Next, we describe the implementation of a prototype on-line change system for programs written in C. The prototype is for changes in which functions are considered as the units of change. It is based on the sufficient conditions developed earlier for this and is implemented using state transfer between processes. We describe some experiments to evaluate the performance of the system and an experiment to determine the usefulness of the sufficient conditions given earlier in ensuring validity in practical situations. The experiments show that the system performs on-line changes with very little disruption to the performance of the software and that the given conditions can be used successfully in most cases to ensure validity.

Next, we consider on-line changes to object-oriented programs. The state is modelled as a directed graph in which vertices represent objects and edges represent the references to the objects. *Data restructuring* forms an important part of on-line changes to object-oriented programs. We formalize this concept in terms of our model of state and use it to describe a class of state mappings which would be typically used in practice. For this class of state mappings, we develop sufficient conditions for validity of on-line changes in this model. Finally, we discuss issues in implementing an on-line change system for object-oriented languages.

Finally, we consider on-line changes to distributed programs. For this purpose, the basic framework is extended to allow multiple concurrently running processes. A general message passing based system model is considered in which communication can be either synchronous or asynchronous. We consider on-line changes in which only one process is changed. For this case, we develop a set of conditions for validity and show how they can be checked using the methods developed for sequential programs. We then discuss on-line changes which involve changes to multiple processes. Finally, we consider a remote procedure call based model and show how the task

of ensuring validity of an on-line change can be simplified for such a “higher level” paradigm of inter-process communication.

# Acknowledgments

It is a pleasure to thank my thesis supervisors Dr. Jalote and Prof. Barua for their constant help, encouragement and support throughout the course of this work. But for their unremitting, insightful criticism of my work, I would still be floundering around with my fanciful notions. Dr. Jalote first introduced me to this problem and motivated me to work on it for which I am grateful to him. I would also like to thank my thesis examiners whose critical evaluation and helpful suggestions have led to the rectification of many mistakes and omissions. Many other people have helped me in this work from time to time. I would especially like to thank Ashish Singhai and Dr. G. Sivakumar for many a stimulating discussion on the subject and Dr. Sanjeev Kumar, Dr. Harish Karnick, Shreesh Jadhav and Rajeev Singh for clearing up occasional doubts and helping out with references. I thank Brajesh Pandey for reading and improving parts of the earlier drafts of this thesis. To all my friends in IIT/K and especially the Hall-4 billiards club and my bridge group, I am grateful for making my stay here one of the most memorable times of my life. Finally, I would like to thank my family for cheerfully bearing with my whims and for letting me make my own decisions.

Deepak Gupta

# Contents

<b>List of Tables</b>	<b>x</b>
<b>List of Figures</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 On-line Change and Fault Tolerance . . . . .	3
1.2 Related Work . . . . .	5
1.2.1 A Dynamic Type Replacement System . . . . .	5
1.2.2 DAS . . . . .	6
1.2.3 DYMOs . . . . .	7
1.2.4 PODUS . . . . .	8
1.2.5 Argus . . . . .	9
1.2.6 Conic . . . . .	10
1.2.7 Durra . . . . .	11
1.2.8 Polyolith . . . . .	11
1.3 Scope and Goals . . . . .	12
<b>2 The Framework</b>	<b>14</b>
<b>3 On-line Change for Procedural Languages</b>	<b>21</b>
3.1 Program Model . . . . .	21
3.2 Undecidability of Validity . . . . .	22
3.3 State Mappings . . . . .	23
3.4 Sufficient Conditions for Validity . . . . .	25

3.5	Algorithm <i>Change</i> . . . . .	29
3.6	An Example . . . . .	32
3.7	A More General Program Model . . . . .	34
3.7.1	Model of State . . . . .	36
3.7.2	Validity of Change . . . . .	37
3.7.3	Change in a Reachable State of $\Pi'$ . . . . .	38
3.7.4	Change in an Unreachable State of $\Pi'$ . . . . .	43
<b>4</b>	<b>Implementation and Experience</b>	<b>47</b>
4.1	The Basic Approach and Design Principles . . . . .	48
4.2	System Design . . . . .	49
4.3	Implementation Issues . . . . .	52
4.3.1	Maintaining Consistency of Pointers . . . . .	52
4.3.2	Deciding the Time of Change . . . . .	53
4.3.3	Handling Open Files . . . . .	54
4.3.4	Initialization Routine . . . . .	54
4.3.5	Adding and Deleting Global Data . . . . .	55
4.3.6	Another Approach to Adding Global Data . . . . .	56
4.3.7	Other Restrictions . . . . .	56
4.4	An Example . . . . .	57
4.5	System Performance . . . . .	60
4.6	Experiments with On-line Change . . . . .	62
4.6.1	Experimental Procedure . . . . .	63
4.6.2	Experimental Data and Observations . . . . .	64
4.6.3	Conclusions . . . . .	65
<b>5</b>	<b>On-line Change for Object Oriented Programs</b>	<b>67</b>
5.1	Program Model . . . . .	70
5.2	Execution Model . . . . .	72
5.3	Model of State . . . . .	75
5.4	State Mappings . . . . .	77
5.5	Ensuring Validity . . . . .	79

5.5.1	Change in a Reachable State of $\Pi'$ . . . . .	80
5.5.2	Change in an Unreachable State of $\Pi'$ . . . . .	85
5.6	Implementation Issues . . . . .	88
5.7	Conclusions . . . . .	92
<b>6</b>	<b>On-line Changes to Distributed Programs</b>	<b>93</b>
6.1	Execution Model and Framework . . . . .	94
6.2	Conditions for Validity . . . . .	97
6.3	Ensuring the Sufficient Conditions . . . . .	99
6.4	Changes to Multiple Processes . . . . .	100
6.5	On-line Change for an RPC Based Program Model . . . . .	102
6.5.1	Systems with Stateless Servers . . . . .	102
6.5.2	Systems with Stateful Servers . . . . .	104
6.6	Conclusions . . . . .	104
<b>7</b>	<b>Conclusions</b>	<b>106</b>
	<b>References</b>	<b>110</b>
<b>A</b>	<b>Algorithm <i>Diff</i></b>	<b>115</b>
<b>B</b>	<b>Correctness of Algorithm <i>Change</i></b>	<b>118</b>

# List of Tables

3.1	Ensuring validity of the on-line change to the factorial program . . . .	34
4.1	Commands of the Modification Shell . . . . .	51
4.2	Observations in the experiments . . . . .	65

# List of Figures

3.1	Algorithm <i>Change</i> . . . . .	31
3.2	The two versions of the factorial program . . . . .	33
3.3	Diagrammatic representation of functional enhancement . . . . .	38
4.1	Major modules of the system . . . . .	50
4.2	Print server main loop . . . . .	58
4.3	Version 2 of function <code>read_command</code> . . . . .	59
4.4	Change initialization routine for print server . . . . .	60
5.1	The <code>main</code> and <code>list</code> classes . . . . .	71
5.2	The <code>listnode</code> class . . . . .	72
5.3	Run time state of the program . . . . .	74
5.4	A state of the example program . . . . .	77
5.5	Changes to the <code>list</code> class . . . . .	83
A.1	Algorithm <i>Diff</i> . . . . .	116

# Chapter 1

## Introduction

Despite advances in software development technology, software systems are, in general, neither error free nor do they satisfy every anticipated need. It is, in fact, well known in software engineering that programs undergo a lot of changes during their lifetime. These changes are often for bug-correction and for adding new functionality which the users of the systems did not initially realize was required and hence did not form a part of the original specifications of the programs. The change is usually effected by stopping the currently running program and then installing the new version. This necessarily implies a denial of service to the users of the software while the switchover is being made. In some cases this downtime may only be irritating to the users of the system, or at worst, cause a monetary loss, for example if the system is for processing bank transactions. For some other systems however, this denial of service may have drastic consequences, for example in an aircraft control system or any other such safety critical system. For such systems, it is clearly desirable to have a modification system which eliminates or reduces the system shutdown time for installing a new version of the software, thereby allowing continuous service to the users of the software. This can be achieved through on-line software change.

A system which supports on-line software version change is one in which the software version can be changed while the system is operating. In other words, the code of the executing software can be changed without shutting down the execution. Clearly, an on-line change can not be made from any arbitrary program to any other

program or installed at any arbitrary time. Understanding what types of changes can be done, when the change can be installed, how to install it etc. are the issues which must be addressed before systems that support on-line changes can be built. In this thesis, we study various aspects of the problem of supporting on-line software change.

On-line software version change is analogous to the on-line replacement of components in hardware but has some important differences. On-line module replacement in hardware is well understood and many commercial fault tolerant systems support it. For example, Stratus systems [Web90] allow changing of printed circuit boards (PCBs) or some modules while the system is running, without shutting down the system. A key feature in these approaches is that they have redundancy in the system and if a board is found faulty, the system is reconfigured to take the board out of operation and make it inactive before removing it.

The goal of on-line software version change is similar in concept. We want to change some software components (modules), and would like to be able to just “pull out” the relevant modules and replace them with the new modules without significantly affecting the system performance. However, there are some key differences in the problem of on-line software version change from its counterpart in hardware. Since we consider a general software system which may not have software redundancy, the situation is different from the hardware systems in which the component being “pulled out” has been effectively removed from the system.

Another difference of on-line software change from on-line hardware change is that the new module may not be exactly the same as the old module. That is, the specifications (both functional and performance) of the new module may be different from those of the old one. This is a fundamental requirement for on-line software change because it is motivated by the need for enhancement or bug removal, both of which require changes in the module.

## 1.1 On-line Change and Fault Tolerance

A fault tolerant system is one in which the failure of some components can be masked such that the system continues to operate in an error-free manner [AL81, Jal94]. Fault tolerance is typically supported by using redundancy. Hardware, software or temporal redundancy may be used to mask failure of components at higher levels [Avis85, Jal94]. In fault tolerant software, the goal is to mask the failure of software components such that the overall software system does not fail. Two major techniques that have emerged for supporting fault tolerant software are the recovery block approach [Ran75] and the N-version programming approach [Avis85]. Both these methods use software redundancy to achieve fault tolerance and thus require multiple versions of the software to be developed which makes them very expensive. Some other techniques that can handle the same types of faults have also emerged e.g. rollback and restart [Gra85], message reordering [FHKR93, WHF93], data re-expression [AK87] and data roll-forward [PJ93]. These techniques detect errors and then use rollback based methods to provide continued service. The basic idea in all these techniques is to try to execute the system again using different inputs.

On-line software version change can help in increasing system availability [GJ93a]. One example of a system that requires a very high availability can be found in the telecommunication domain where a switching system may be required to have a downtime of no more than 2 hours in 40 years. Traditionally, the techniques for enhancing availability have mostly focussed on masking hardware failures and little attention has been given to the effect of software failures on availability. For availability analysis of a system also, typically only hardware failures are considered and the effect of software failures on availability is not taken into account. It is now well known that software fails more often than hardware in most systems. Therefore to get a good idea of the availability of a system, it is important to consider software failures as well. If  $MTTF$  is the mean time to failure of the system and  $MTTR$  is the mean time to repair, then availability of a system is given by

$$\frac{MTTF}{MTTF + MTTR}$$

It is clear that availability can be increased either by increasing the  $MTTF$  or by

reducing the MTTR. In hardware, both approaches have been utilized for increasing the hardware availability. Fault tolerance techniques are used to increase the MTTF, and fast replacement schemes are used to minimize MTTR. Some commercial fault tolerant systems have elaborate methods for replacing failed components, in which the non-failed part of the system performs the diagnosis and informs the service department about the failed component.

The focus of techniques to support fault tolerant software is on increasing the MTTF. The focus of the quality assurance methods is the same. In contrast, the emphasis of on-line software version change is on reducing the MTTR of software. The “repair” time in the case of a software failure is the time taken to detect and remove the fault and then reinstall the system. This repair time can be very large. If the failure is not fatal, detection and removal of the fault can be done off-line and an on-line version change system can significantly reduce the reinstallation time, thus reducing the mean time to repair and increasing the availability considerably.

For example, consider a system with MTTF of 24 hours, mean hardware repair time of 2 hours and mean software repair time of 30 minutes. If 70 percent of the failures are software failures, then the availability of the system will be 0.962. If by using on-line software version change, we can reduce the software repair time from 30 minutes to 1 minute the availability will increase to 0.975. And this availability increase is relatively inexpensive as it does not require any redundancy in the hardware or the application software.

An on-line software version change system can also be used in conjunction with fault tolerance techniques for providing continuous operation in a cost effective manner. As mentioned earlier, there are some rollback and retry based methods that can be used to mask software failures. Though these methods are cost effective, they do not remove the software fault that caused the failure. These faults can cause failures again in the future and may prevent a successful recovery. To overcome this drawback, on-line software version change can be used at the time of recovery. The rollback technique masks the current error while on-line software replacement removes the software fault by replacing the faulty software with a fault free version developed off-line. Thus, on-line software replacement can be effectively combined

with some of the software fault tolerance techniques for providing a high availability at a low cost.

## 1.2 Related Work

In this section, we briefly review some of the previous work in the area of on-line software modification. A comprehensive survey can be found in [SF93].

### 1.2.1 A Dynamic Type Replacement System

Fabry described a system in which implementations of abstract data types (called modules) can be changed on the fly [Fab76]. Several processes may access the same module and a module may manage permanent data local to one process or shared by several processes. Fabry only considered the case in which the interface and the semantics of the module being replaced do not change across versions. The system was based on capability based addressing [Fab74]. The call to a module is made by specifying a *capability* for the module in the call instruction. This capability points to an indirect word containing an instruction to jump to the actual address in memory of the module's code. The modification to a module is made by substituting the capability for the module by a new one which points to a new indirect word containing an instruction to jump to the address of the new version of the module. This substitution is effected by using the capability revocation mechanism which ensures that the meaning of all copies of the capability (held by different processes) is changed simultaneously.

Changing the implementation of an abstract data type may require a restructuring of the permanent data that it manages. In Fabry's system, this is not done at the time when the change is installed but when an instance of the data structure is first used after the change. This leads to a significant reduction in the time taken for the change since restructuring of all instances of a data type may take a long time. Further, some of these instances may never be used at all. Version numbers are used to detect obsolete data for this purpose. Thus the entry code for a module checks if the version number is stored in the data structure is less than the version number

of the code and if so, it calls a conversion routine to do the necessary restructuring. Since the restructuring is done on demand, an instance of a data type may be several versions out of date when used. To take care of this problem, the conversion routine calls the previous conversion routine if necessary.

Since several processes may simultaneously access the same module and there is assumed to be no mechanism to detect which processes are accessing a given module at a given time, an elaborate locking mechanism is used to synchronize the users of a module and to ensure that the modification is also properly synchronized.

### 1.2.2 DAS

DAS (Dynamically Alterable System) [GIL78] is an experimental operating system which provides support for dynamic updating of programs by allowing a module to be replaced by a new version having the same interface. The system is designed in such a way that dynamic modifications are possible in all system components except in the kernel which contains part of the modification facility. The facility is conceptually similar to dynamic type replacement of Fabry's system. Dynamic updating is implemented by the mechanism of "replugging" which is based on the addressing scheme of the system. Each module resides in a different address space which consists of several segments. The information used to keep track of each module is a linked list containing an entry for each segment. A call to a procedure of a different module is achieved by an address space transition performed by a special CALL instruction which replaces some segments in the virtual address space by some other segments. Replugging is performed by changing the links in the descriptor chain of the modified module. Data restructuring is done on request rather than on demand. Again, since multiple processes are allowed to access the same modules, a locking scheme is used for synchronization among the users and the "repluggers" of a module.

The authors also gave a general scheme for data restructuring which does not require a conversion algorithm for every possible pair of implementations of an abstract data type. The basic idea is that the data type is augmented with two operations *in* and *out* which are responsible for piecewise "filling" and "emptying"

the information in the data structure. Using these operations, it is possible to use the same algorithm for every possible pair of implementations of a data type since the representation details of the old and the new implementations are encoded in the *out* operation of the old implementation and the *in* operation of the new implementation respectively. The functional behavior of these operations is described by the common specification of all versions.

### 1.2.3 DYMOS

Insup Lee, in his doctoral dissertation, presented a dynamic modification system called DYMOS [Lee83]. DYMOS is a fully integrated environment for software development and program updating. It supports programs written in the language StarMod. StarMod is a concurrent language which also supports data abstraction and provides synchronization using a monitor like construct. The DYMOS environment includes a command interpreter, a source code management system, a StarMod compiler, an editor and a run-time environment. The system allows individual procedures of a program to be changed. Because of the integrated nature of the system, the source code, the object code and the symbol table etc. are always available and are used by the system to aid on-line changes. The system tries to ensure that the program image in the memory always corresponds to the latest version of the source code.

Lee also gave a procedure for partitioning a change (i.e. the set of procedures changed across versions) into a sequence of smaller changes. The decomposition is done in such a way that the program behaves “acceptably” after each change. In other words, each of the intermediate programs obtained after any of the smaller changes should be “functionally consistent”. The criteria for doing this are based on the specifications of the changed procedures. The basic idea is that if the old version of a procedure  $A$  “depends on” the old version of a procedure  $B$  but can not use the new version of  $B$  instead without violating its specifications, then  $A$  should be updated before or with  $B$  so that the old version of  $A$  never calls the new version of  $B$ . Similarly, if the new version of  $A$  can not use the old version of  $B$  instead of its new version, then  $B$  should be updated before or with  $A$ . Based on these criteria,

the set of changed procedures is partitioned into a sequence of pairwise disjoint subsets such that the intermediate programs after each of these “sub-changes” are functionally consistent.

### 1.2.4 PODUS

PODUS (Procedure-Oriented Dynamic Updating System) is an on-line software version change system developed at the University of Michigan and later enhanced at Bellcore [SF89a, FS91]. As its name suggests, PODUS supports dynamic modification to procedural programs. Procedures are considered the units of change. Procedures are updated one by one till the change is complete. The system allows changes in the interfaces of procedures using what are known as “inter-procedures” and also allows data restructuring to be specified using “mprocedures”. The system has two main components: the updating shell (*ush*) and the program update process (*pup*). The *ush* is the user-interface to the system and provides commands for loading, running and dynamically updating programs. The *ush* sends these requests to the *pup* in whose address space, the user program is run. The *ush* and the *pup* communicate using internet domain sockets and thus need not reside on the same physical computer. PODUS is based on a very large sparse address space architectural model. The address space is partitioned into a number of version spaces using a “version id” in the address. Each version space holds all the data and code of a program version besides a binding table. The required large sparse address space is not directly provided by many architectures but can be simulated by using extra addressing information in machine registers etc.

PODUS also supports dynamic changes to distributed programs in which processes communicate through the remote procedure call (RPC) mechanism [SF89b]. An enhanced version, called REV PODUS also supports the remote evaluation mechanism of inter-process communication [Seg91].

In PODUS, a procedure is updated only when it is inactive. A procedure is defined to be active at a given time if it is not on the run-stack and none of the procedures that its new version may directly or indirectly call is on the run-time stack. This implies that at no time, the new version of a procedure can call the old

version of another procedure. Clearly, this condition is similar to Lee's conditions for partitioning the set of changes into a sequence of smaller changed but with an added assumption that the old version of a procedure can always use the new version of another procedure instead of its old one and the new version of a procedure can never use the old version of another procedure instead of its old one. However, PODUS also allows semantic dependencies to be specified between procedures which allow two or more procedures to be updated at the same time. These dependencies can not be inferred from the syntax of the program and have to be specified by the user. However, no guidelines are given for determining them.

### 1.2.5 Argus

Argus is an integrated programming language and system designed to support construction of well-designed distributed software [Lis88]. Argus is based on the CLU programming language and provides support for atomic transactions and crash recovery. An Argus program consists of a set of servers called *guardians*. A guardian is similar in nature to an abstract data type. Communication between guardians is done by means of a remote procedure call like mechanism. In [Blo83] a dynamic module replacement facility for Argus has been described. The unit of replacement in this system is a collection of guardians, called a *subsystem*. The condition for an acceptable on-line replacement of a subsystem requires that the new instance only generate those event sequences (called *futures*) which would have been permitted by the replaced abstraction in the state in which the replacement occurs. Based on this condition, the required relationship between the abstract specifications of the new and the old subsystem versions is given as also the restrictions on the state in which the change takes place. The main weakness of this definition is that it constrains the new version of a replaced subsystem to be closely related in behavior to the old one. Thus it may not be possible to handle changes which are required because of bug fixing. It is, sometimes, not even possible to replace a subsystem by a new implementation of the same abstract specifications. The main drawback of the replacement system is that it requires the Argus crash recovery facilities in order to work properly and may not be adaptable to other systems. Further, a subsystem

may be too large a unit of change in some applications.

### 1.2.6 Conic

Conic is a system developed at the Imperial College for supporting dynamic reconfiguration of programs [MKS89]. It provides a language and a run-time environment for constructing distributed programs. A program in Conic consists of a set of modules each having a number of entry and exit ports. Modules communicate with each other using these ports which serve as unidirectional links. System configuration is specified separately in a language which allows creation of instances of modules and linking of entry ports of these instances to exit ports of other instances. Thus, Conic modules do not communicate by naming each other but by naming the ports.

During system execution, the configuration of the system can be changed by providing a configuration change specification [KM85]. This specification can ask for creation of new instances of modules, deletion of old instances and creation and deletion of links between ports of various modules. The *configuration manager* translates these requests to change the system into commands to the distributed operating system to execute reconfiguration operations.

In [KM90], Kramer and Magee gave a formal basis for dynamic reconfiguration of distributed programs. They considered the interactions between processes (called nodes) to consist of *transactions*. A transaction is an exchange of information between two nodes of the system. It is initiated by one of the nodes and consists of a sequence of one or more message exchanges between the nodes. A node is defined to be *passive* if it is not currently engaged in a transaction that it initiated and will not initiate any new transactions. A node is defined to be *quiescent* if it is passive, is not currently servicing any transaction and no transactions have been or will be initiated by other nodes which require service from this node. It was claimed that a dynamic reconfiguration will leave the system in a “consistent” state if all the involved processes are quiescent at the time of reconfiguration. No justification was however given for the claim and the issue was not considered in detail. The work concentrated mainly on showing how the quiescent state can be reached with the cooperation of some nodes.

### 1.2.7 Durra

Durra is a language designed to support the development of distributed applications on heterogeneous machine networks [BDW90]. Like Conic, it separates the specification of the behavior of the system components from that of the system structure. It also provides support for run-time reconfiguration of the system and the facilities provided for this are very similar to those in Conic. However, the motivation for providing such capabilities in Durra is not to support changes in the programs but to provide fault tolerance (against such faults as processor and link failure etc.) The consequence of this is that in Durra, all reconfigurations must be anticipated and specified in advance.

### 1.2.8 Polyolith

Like Conic and Durra, Polyolith is a system which aids development of distributed applications by allowing the structure of the application to be specified separately instead of being hard-coded in the program [Pur90, PJ89]. It provides a specification language (MIL) for specifying the structure and a *software bus* which provides the run-time support. Dynamic reconfiguration facilities of the system are described in [PH91]. Besides changes in the structure of the programs, Polyolith also allows changes in geometry and individual module implementations. Purely structural changes (adding or deleting modules and changing links between them) can be done without any support from the application itself but to replace a module by a new implementation or to relocate a module to a different machine, the system needs the cooperation of the involved module since the state must be captured and then restored in the reincarnation of the module. For this purpose, the module must provide two operations: *encode* and *decode*. The *encode* operation is responsible for encoding the present state of the module in an abstract form. The *decode* operation uses this abstraction of the state to restore the concrete state in the newly created module. The abstraction of state for transfer ensures that the new and the old modules need not reside on architecturally similar machines. To enlist the module's cooperation in saving the state, the system sends it a special command which induces it to encode the state. However, the module need not immediately stop and encode the state but

can continue execution till it deems its state “consistent” for a transfer. In effect, the actual time of replacement is left to the program. The system does not disallow replacing a module by a new one which might have different semantics from the old one but no guarantees can be given, in this case, about the system behavior after the change. Presumably, the *decode* operation of the new version must make some sense out of the old version’s state and ensure that the overall state of the system is consistent after the change. The problem of how this can be done is not, however, considered.

In [HWP92], a tool to aid in dynamic reconfiguration without module reconfiguration was described. The following general characteristics of modules which can be reconfigured in this manner were identified.

- The module’s state can be safely discarded during reconfiguration.
- The module does not require any special initialization when created dynamically.
- There is no synchronization between the module and its neighbors.

In [HP93], a tool was described to capture and restore a module’s state automatically. The approach used was to preprocess the module code to insert statements to capture and restore the state. However, since the saved state contains values of variables of the program and also explicit reference to control points within the program, it is not suitable if the module is replaced by a new version (since the new version may have different variable names and the control points of the old version may not have “equivalents” in the new version) even if the two versions have the same functional specifications.

### 1.3 Scope and Goals

As the above survey shows, the focus of most of the work done in this area has been to implement systems using which components of a software system can be changed on-line. However, few efforts have been made to study the nature of the problem in a formal framework and the limitations of such systems. Specifically, the behavior

of a program after an on-line change has not been studied. It can be seen that this behavior can be “erroneous” even if both the programs are correct. One must give a definition of “acceptable” behavior after the change and then give conditions under which such behavior will be observed.

The main goal of our research is to study the “validity” of an on-line change. We give a formal framework for studying on-line changes and formalize the notion of validity of an on-line change. We show that it is, in general, undecidable whether a given on-line change is valid. This shows that building a system which will automatically ensure the validity of the on-line changes which it installs is impracticable and that some user-assistance is indispensable for ensuring validity. Since the undecidability of the problem rules out computable necessary and sufficient conditions for the validity of on-line changes, we develop computable sufficient conditions for ensuring validity in three different programming models — the procedural language model, the object-oriented programming model and the distributed system model. We also consider some implementation issues for these models and for the procedural model, describe a simple prototype implementation and some experiments with it.

The rest of this thesis is organized as follows. In Chapter 2, we give a general framework for studying on-line changes. In Chapter 3, we consider the validity of on-line changes to programs written in a sequential, imperative language such as Pascal. We show that the problem is, in general, undecidable and then develop sufficient conditions for validity, first for a simplified program model devoid of functions and procedures and then for more realistic languages with procedures. Chapter 4 describes a prototype implementation of an on-line software version change system for procedural languages. It also describes some experiments conducted to evaluate the performance of the system and to investigate some engineering issues involved in on-line change of programs. In Chapter 5, we consider an object-oriented program model and give sufficient conditions for validity in this model. Distributed systems are considered in Chapter 6. In this chapter, we give sufficient conditions for validity first for unrestricted message passing based systems and then for RPC based systems. The conclusions and scope for further work are presented in Chapter 7.

# Chapter 2

## The Framework

In this chapter, we present a formal framework for studying on-line changes. The definitions and notations given here will be used more or less unchanged in the following chapters. For now, we focus on sequential systems where a single process executes the program. We will later extend the framework for distributed systems.

Any formal framework for on-line change has to deal with programs, processes and process states. What exactly constitutes a program is defined by the particular program model considered. For example, for procedural languages, a program may be modelled as a sequence of statements.

A *process* is a program in execution. It has two components — the code of the program being executed and a *state*. Typically, the code of a process remains the same through out the lifetime of the process and the state changes according to some state transition function which specifies the next state depending on the current state and the program being executed. This function forms the *execution model*. For each program  $\Pi$ , a unique *initial state*  $s_{\Pi_0}$  is defined which is typically the state of a process executing  $\Pi$  at the very beginning of its lifetime. Thus, given the program being executed by a process, a state is the complete characterization of a point in the lifetime of the process. What exactly constitutes a state and what is the initial state for a given program is specified by the program model being considered. In this thesis, we have modelled a state as a function, a graph etc. for different models. Although, the code executed by a process typically remains the same, it need not be

so. Thus we treat a process as an independent entity as in most operating systems and the code of the program being executed by it as a part of it which can be changed during execution if necessary.

**Definition 2.1** A state  $s$  is said to be a *reachable* state of a program  $\Pi$  if and only if a process executing  $\Pi$  from its initial state  $s_{\Pi_0}$  can reach  $s$  at some time for some inputs.

Thus when started from its initial state, a process executing a given program stays in one of the reachable states of the program. However, we do not make the restriction that a process must be started from the initial state of the program it is executing or that it should always be in a reachable state of this program.

As the state is the complete characterization of a process, a process need not be started with the initial state of the program that it is to execute but can be started from any state. If a process executing the program  $\Pi$  is started from a reachable state of  $\Pi$ , then the behavior of the process is identical to the behavior if the state was reached by the process starting from  $s_{\Pi_0}$ . That is, the behavior of the process starting from a reachable state is independent of how that reachable state was obtained. Specifically, the reachable state could have been created by another process.

Program models typically allow input/output instructions in the programs. The execution of the input instruction causes some data to be read (from some input device), which, in turn, changes the state. The execution of an output instruction causes some data to be written to some output device. These instructions are thus the means of interaction between the program and its environment.

The on-line change problem relates to both the process and the program. The basic idea is that we want to change the program  $\Pi$  being executed by a process, while it is executing, by another program  $\Pi'$ . We assume that both  $\Pi$  and  $\Pi'$  are “correct” programs in the sense that by themselves they can execute properly and produce expected results for the inputs provided to them. That is, the behavior of both  $\Pi$  and  $\Pi'$  is well understood for their respective input domains. The problem we focus on is the behavior of a process for which during the execution,  $\Pi$  is changed to  $\Pi'$ .

While performing an on-line program version change, it is often desirable and may indeed be necessary to map the state of the process for various reasons: for example, to initialize a variable added in the new version, to change the type of a variable, and so on. In our model, we allow this mapping, in its full generality, by requiring a state mapping function which is used to map the state at the time of change. The case when no mapping needs to be done can be handled by using the identity state mapping. We now define an on-line program version change.

**Definition 2.2** An on-line change from program  $\Pi$  to  $\Pi'$  at time  $t$  using the state mapping  $\mathcal{S}$ , in a process  $P$  (executing  $\Pi$ ) is equivalent to the following sequence of steps:

1.  $P$  is stopped at time  $t$  in state  $s$  (say).
2. The code of  $P$  (which, till now, was the program  $\Pi$ ) is replaced by the program  $\Pi'$ , its state is mapped by  $\mathcal{S}$  and  $P$  is then continued (from state  $\mathcal{S}(s)$  and with code of  $\Pi'$ ).

The above definition shows that we have modelled an on-line change as an instantaneous rather than an incremental process. This leads to a cleaner and more general formalization. Incremental changes can be modelled as a sequence of instantaneous changes.

It is clear that any arbitrary on-line change will not produce meaningful and acceptable results. First, the process may not be able to execute after its program has been changed. Even if the process continues to execute, its behavior may not be “acceptable”. After all, the basic goal of an on-line change is to change the existing behavior of the system to some other (acceptable) behavior. Before we can make any claims about the on-line change, we must clearly specify the expected behavior after the change. Clearly, the desired behavior is that the process behaves as it would with the new program. However, since after the change, the new program starts getting executed from a state which is typically not its starting state, expecting exactly the same behavior is unrealistic. We propose the following definition of *validity* of an on-line change.

**Definition 2.3** An on-line change in the process  $P$  from  $\Pi$  to  $\Pi'$  at time  $t$  (in state  $s$ ) and using the state mapping  $\mathcal{S}$  is *valid* if after the change,  $P$  is guaranteed to reach a reachable state of  $\Pi'$  in a finite amount of time.

Thus an on-line change is valid, if after a certain “transition period” after the change, the process starts behaving as if it had been executing the newer version of the program since the beginning from its initial state. The process may behave in an arbitrary fashion during the transition time when it is yet to reach a reachable state of the newer version.

It can be seen that validity of an on-line change depends on the two versions  $\Pi$  and  $\Pi'$  of the program, the time  $t$  of change (or equivalently, the state  $s$  in which the change is made) and the state mapping  $\mathcal{S}$  used. Clearly a meaningful state mapping can only be specified by the user based on his semantic knowledge of the two versions of the program. For example, if a variable  $x$  of  $\Pi$  has been renamed  $y$  in  $\Pi'$ , this information is available only to the user. Thus we assume that the state mapping  $\mathcal{S}$  is given to us and thus, given  $\Pi$ ,  $\Pi'$  and  $\mathcal{S}$ , our task becomes limited to giving conditions on the time of change so that validity is guaranteed.

So far we have not put any restrictions on the nature of or the relationship between  $\Pi$  and  $\Pi'$ . However if  $\Pi$  and  $\Pi'$  are two completely different and unrelated programs, it is clear that there may not be any state in which an on-line change from  $\Pi$  to  $\Pi'$  is guaranteed to be valid. Thus while giving conditions for validity, we will usually assume certain properties of the two programs  $\Pi$  and  $\Pi'$ . We have chosen not to make these assumptions in the definition of an on-line change itself because these assumptions will, in general, depend on the particular program model being considered and including them in the basic framework will reduce its generality.

Note that the definition of validity allows the reachable state of  $\Pi'$  finally reached to be *any* reachable state of the  $\Pi'$ . In particular, this state may not be obtainable through the same (or a “corresponding”) sequence of inputs which led to the state  $s$  of  $\Pi$  in which the change was made. This will however happen only if the state mapping used is such that this correspondence is not preserved. Typically, the mapping will map a state to a “similar” state which is likely to have been obtained through “similar” inputs. We will also usually want the new version to start “where

the old one left off”. These requirements must be captured by the state mapping being used. Again, in the interest of generality, we make such assumptions about the state mapping when we discuss conditions for validity rather than in the general definition of an on-line change.

Another important issue is that after the change, the inputs given must be suitable for the new version of the program. If this is not done, a program crash may result even if the change was valid. In the case of interactive inputs, the user of the software is responsible for giving “correct” inputs after the change. If the inputs, however, come from a file on permanent storage, then the contents of the file may also need to be changed at the time of change. In other words,  $\Pi'$  may work with a different set of files. We consider our requirement that  $\Pi'$  is independently correct in a general sense implying that it behaves in an expected fashion if the inputs given are within the acceptable domain i.e. the inputs files are appropriate for it. However, even if the inputs are such that  $\Pi'$  behaves correctly, it may still be necessary to map the current offset(s) within the file(s) for  $\Pi$  to within files for  $\Pi'$ . Typically, this mapping will be performed, if needed, by the state mapping function  $\mathcal{S}$ .

It is important to note that we are not concerned with the “correctness” of a change i.e. whether after applying a certain change to a program (for example, replacing some procedures by new procedures), the new program obtained is a “correct” program. In this work, we assume that both the new and the old versions of the program are independently “correct”. Correctness of a change is a “static” notion and can be settled by examining only the text of the changed program (by a human if not automatically.) The validity of an on-line change is, however, related to the dynamics of the change and depends not only on the program before and after the change but also on the timing of the change and on the state mapping used at the time of change. The process in which an on-line change has been effected executes the old version of the program for some time and the new version for the rest of its duration. Thus, its overall behavior (as its user sees it) may not resemble the behavior expected of either of the two versions. The question that validity of the on-line change asks is whether the process will ever, after the change, start behaving like the new version of the program is expected to. A simple example can be used to

illustrate the difference between the notions of correctness of a change and of validity of an on-line change. Consider a program which uses a stack as a data structure implemented by an array. Suppose that in the new version of the program, the stack is implemented as a linked list but otherwise the new version is identical to the old one. Clearly, the new version of the program has the same abstract specifications as the old one and is “correct”. However, an on-line change from the old to the new version will only be valid if a state mapping which maps the array representation of the stack to an equivalent linked list is used at the time of change. In this thesis, we focus on the validity of changes and assume that the correctness of the change has already been established by the user.

It would also, at this point, be instructive to compare our framework and goals with the work of Lee [Lee83] and that of Segal and Frieder [SF89a]. Lee gave a procedure for partitioning a given (correct) change into a sequence of smaller changes each of which is such that the intermediate programs are “functionally consistent” i.e. satisfy the specifications of either the old or the new version of the program. The criteria for partition are based on the specifications of the old and new versions of the changed procedures. The sequence in which procedures should be updated as given by Segal and Frieder are based on similar conditions. In this thesis, however, we are addressing an orthogonal question. We model the on-line change as an instantaneous process and try to predict the observed behavior after it. Thus each of the sub-changes obtained by partitioning the overall change using the approach of Lee or of Segal and Frieder can be considered as a single change in our framework. In this respect, this work is complementary to the previous work.

As can be seen, we have used an *operational semantics* model [Rey72] for modelling on-line changes to programs. Such a model has been preferred over other possibilities for the following reasons.

**Generality** The model can be easily applied to different programming paradigms by appropriately defining the state and the transition function. It is worth noting that we have purposely defined our notion of validity without assuming any programming language model. In the following chapters, we will use the same general definitions (modified slightly for the distributed case) for three

different programming models — the sequential procedure based programming model, the object-oriented model and the distributed model.

**Closeness to implementation** The model is closely related to the way many programming languages are usually implemented (i.e., with a process state and instructions which modify the state in well defined ways) and is therefore easier to follow (in our opinion). The definition of an on-line change (Definition 2.2) is very similar to how it might be implemented in practice. Due to this closeness, we are able to view the results in terms of the actual source code of the program.

Conceivably other models such as denotational semantics [Sto77, Sch86], Vienna Definition Method [BJ82] etc., can be used to model on-line changes. However it is not clear if these other models can be used for different programming paradigms, particularly distributed programs, and whether the dynamics of on-line changes can be modelled as easily as with a general state based approach.

The above definitions will be slightly modified to handle distributed programs in Chapter 6. In particular, a program execution will be defined to consist of several processes running concurrently, distinction will be made between system and process states and the definition of on-line change will be modified to allow only a subset of the processes to stop executing during the change.

## Chapter 3

# On-line Change for Procedural Languages

In this chapter, we consider on-line changes to programs written in a sequential imperative language such as Pascal. We first consider a simple program model in which the programming language has no procedures and functions. For this model, we show that the problem of determining validity of change is undecidable. We then proceed to develop sufficient conditions for validity. Finally we extend the model to consider a more realistic programming language with procedures and functions and develop sufficient conditions for this model.

### 3.1 Program Model

We first consider a simple program model. The language we consider is like any imperative language (like C, Pascal etc.) except that procedures and functions are not allowed. Thus a program in this hypothetical language consists of a set of typed variables and a sequence of statements which can be assignment, conditional (if-then or if-then-else), looping (while-do) or read/write statements. We denote the set of variables of a program  $\Pi$  by  $V(\Pi)$ . For simplicity, we assume that input from files is not allowed.

For this program model, a *state* of a process executing the program  $\Pi$  is a function

which maps variables in  $V(\Pi)$  and the *program counter* (PC) to values in appropriate domains. Some of the variables (but not PC) may be mapped to the undefined value  $\perp$ . We assume that if a variable has the value  $\perp$  in a state, then using the value of this variable in this state has unpredictable results. The initial state  $s_{\Pi_0}$  of a program  $\Pi$  is the state which maps all the variables in  $V(\Pi)$  to  $\perp$  and PC to the control point just before the first statement of  $\Pi$ . The meaning of the different kinds of statements is as in conventional languages such as Pascal and therefore we do not formally define the execution model.

## 3.2 Undecidability of Validity

Before attempting to find general conditions for validity, we show an important property regarding the decidability of the on-line version change problem in the above specified program model. Specifically we show that given arbitrary programs  $\Pi$  and  $\Pi'$ , state mapping  $\mathcal{S}$  and a reachable state  $s$  of  $\Pi$ , determining whether an on-line change from  $\Pi$  to  $\Pi'$  using state mapping  $\mathcal{S}$  in state  $s$  is valid or not is undecidable.<sup>1</sup>

For proving the result, we first prove the undecidability of a modified halting problem which we will subsequently reduce to the problem of determining validity. For a program  $\Pi$ , let  $C_{\Pi}$  denote the control point just after the last statement of  $\Pi$ .

**Lemma 3.1** *For an arbitrary program  $\Pi$  and state  $s$  such that  $s(PC) = C_{\Pi}$ , it is undecidable whether  $s$  is a reachable state of  $\Pi$ .*

**Proof:** Suppose the above was decidable. Then we can decide whether or not an arbitrary program  $\Pi'$  halts on some input or not as follows. Construct a program  $\Pi$  which first executes  $\Pi'$  and if it terminates, assigns the value 0 (say) to all the variables and then terminates. Clearly,  $\Pi'$  halts on some input if and only if the state  $s$  such that  $s(x) = 0$  for all  $x \in V(\Pi)$  and  $s(PC) = C_{\Pi}$  is reachable for  $\Pi$ . But it is undecidable whether or not a program halts on some input. Hence the given problem must also be undecidable. ■

---

<sup>1</sup>Henceforth we will always represent the time of change by the state in which the change is made.

**Theorem 3.1** *For arbitrary  $\Pi$ ,  $\Pi'$ ,  $\mathcal{S}$  and  $s$ , where  $s$  is a reachable state of  $\Pi$ , it is not decidable if an on-line change from  $\Pi$  to  $\Pi'$  in state  $s$  using state mapping  $\mathcal{S}$  is valid or not.*

**Proof:** Suppose this problem was decidable. Then for any  $\Pi'$  and any state  $s'$ , it is decidable as to whether or not  $s'$  always leads to a reachable state of  $\Pi'$ , since an appropriate  $\Pi$ , its reachable state  $s$  and a mapping  $\mathcal{S}$  can always be constructed such that  $\mathcal{S}(s) = s'$ . Now consider any state  $s'$  such that  $s'(\text{PC}) = C_{\Pi'}$ . By definition,  $s'$  never leads to any other state (when  $\Pi'$  executes from  $s'$ ). By assumption it is decidable whether or not  $s'$  always leads to a reachable state of  $\Pi'$ . Now if  $s'$  leads to a reachable state then  $s'$  itself must be reachable and if  $s'$  does not lead to a reachable state of  $\Pi'$ , it itself is not reachable. Thus for an arbitrary program  $\Pi'$  and state  $s'$  such that  $s'(\text{PC}) = C_{\Pi'}$ , we can decide whether or not  $s'$  is a reachable state of  $\Pi'$ . This contradicts Lemma 3.1. ■

Undecidability of the problem has important consequences. It means that we can not develop a general purpose algorithm which given  $\Pi$ ,  $\Pi'$ ,  $\mathcal{S}$  and  $s$  will tell us whether or not an on-line change with these parameters is valid. In terms of finding conditions for validity, it means that we can not hope to obtain computable necessary and sufficient conditions for validity. We will therefore concentrate on finding computable sufficient conditions for validity.

Note that undecidability of the validity in our restricted program model implies undecidability in more general program models such as for languages with procedures and functions such as Pascal or C, or for object-oriented and distributed program models.

### 3.3 State Mappings

In order to simplify the problem, we now define a class of simple state mapping functions to which we shall restrict ourselves. The functions in this class map the PC value of a state  $s$  independent of the values of variables in  $s$ . Further, the value of a variable in the mapped state is either  $\perp$  or always equal to the value of a variable

in the input state. Formally, for a variable  $y$  of  $\Pi'$ , either for all  $s$ ,  $(\mathcal{S}(s))(y) = \perp$  or there exists a variable  $x$  of  $\Pi$  such that for all  $s$ ,  $(\mathcal{S}(s))(y) = s(x)$ .

It should be seen that the above condition is sufficient to handle most mappings of state that one would desire in practice. It can easily model the identity mapping where the names and values of variables remain the same. It also takes care of change of variable names and addition of new variables (by assigning them the value  $\perp$  in the mapped state). It can not however take care of mappings in which the value of a variable in the mapped state is an arbitrary function of the values of some variables in the state before the mapping. It is clear that such mappings are difficult to analyze automatically because such an analysis would involve semantic knowledge of the domain from which the values of variables are taken. It also can not handle the state mapping required for the stack example of the previous chapter. Such type changes are, however, more easily handled in an object oriented model and will be dealt with in Chapter 5.

A mapping  $\mathcal{S}$  in this class can be specified as an ordered pair  $(\mathcal{C}, \mathcal{V})$  where  $\mathcal{C}$  is a partial one-to-one (but not necessarily onto) function which maps possible PC values of  $\Pi$  to those of  $\Pi'$  and  $\mathcal{V}$  is a partial one-to-one (but not necessarily onto) function which maps variable names of  $\Pi$  to those of  $\Pi'$ . The state mapping function  $\mathcal{S} = (\mathcal{C}, \mathcal{V})$  can be formally defined as

$$\mathcal{S}(s) = \begin{cases} \left( \begin{array}{l} s'(\text{PC}) = \mathcal{C}(s(\text{PC})) \\ s' \text{ s.t. } \\ s'(x) = \begin{cases} s(\mathcal{V}^{-1}(x)) & \text{if } \mathcal{V}^{-1}(x) \text{ exists} \\ \perp & \text{otherwise} \end{cases} \end{array} \right) \text{ if } \mathcal{C}(s(\text{PC})) \text{ is defined} \\ \text{undefined otherwise} \end{cases}$$

Using an argument similar to the one in the proof of Theorem 3.1, it can be easily shown that even under these restrictions on the state mapping, the problem of determining validity is not decidable. We could, of course, have made the problem decidable by using a still smaller set of permissible state mappings (e.g. by allowing only the mapping which maps every state to  $s_{\Pi'_0}$ , the initial state of  $\Pi'$ ) but then many state mappings which would be desired in practice would not be allowed and therefore the results would not be useful in practical situations.

Note that the state mapping need not be defined for all reachable states of the program  $\Pi$ . Clearly the on-line change can not be made in any state for which the state mapping is not defined. Thus the state mapping also puts a restriction on the time of change.

### 3.4 Sufficient Conditions for Validity

Given programs  $\Pi$  and  $\Pi'$  and a state mapping  $\mathcal{S}$ , our task now is to find sufficient conditions which guarantee the validity of an on-line change from  $\Pi$  to  $\Pi'$  using mapping  $\mathcal{S}$  in any state  $s$  which satisfies these conditions. For pragmatic reasons, we want these conditions to only specify restrictions on the program counter value in  $s$  (i.e.  $s(PC)$ ). This is because if restrictions are imposed on the values of variables as well then in an implementation it would be very difficult to check these conditions. The on-line change system has to wait till these conditions are satisfied by the state of the process before proceeding with the installation of the change. This can not be done by checking the conditions at random points in time or at regular intervals since in that case, the system may never be able to stop the process in a state which satisfies the required conditions. The only possible solution would be to check these conditions after the execution of each program statement which would make the overhead of on-line change unacceptably large. On the other hand, if the only restriction on the state  $s$  is that its PC value must be one of some specified control points of  $\Pi$ , the on-line change system can simply wait till execution reaches one of these control points by using a mechanism similar to breakpoints in debuggers. It can be shown however, that even finding *all* such control points is undecidable. The best that we can do, therefore, is to give conditions which enable us to compute *some* such control points.

Intuitively, it can be seen that a change is valid if from the control point from which execution resumes after the change, all the variables which “have been affected by the change” are guaranteed to be redefined before any use. The following theorem proves this.

**Theorem 3.2** *An on-line change from  $\Pi$  to  $\Pi'$  in a state  $s$  for which  $\mathcal{C}(s(PC))$  is defined, using the state mapping  $\mathcal{S} = (\mathcal{C}, \mathcal{V})$  is valid if, there exists a reachable state  $s''$  of  $\Pi'$  such that  $s''(PC) = C'$ , and, for all  $y$  in  $V(\Pi')$ ,  $y$  will either always be defined before use when  $\Pi'$  starts executing from  $s'$  or  $s'(y) = s''(y)$  where  $s' = (\mathcal{C}, \mathcal{V})(s)$  and  $C' = s'(PC)$ .*

**Proof:** If the stated conditions are satisfied, then for any given input sequence, there exists a state  $s'''$  of  $\Pi'$  which is reached from both  $s'$  and  $s''$ .<sup>2</sup> This state is reached from  $s'$  or  $s''$  when all the variables which had different values in  $s'$  and  $s''$  have been redefined. Since  $s''$  is reachable and  $s'''$  is reachable from  $s''$ ,  $s'''$  is also reachable and hence the change is valid. ■

The theorem states that for validity, the mapped state  $s'$  should be the same as some reachable state  $s''$  of  $\Pi'$  (with  $s''(PC) = s'(PC)$ ) except possibly in the values of variables which get redefined before use when  $\Pi'$  executes from  $s'$ . Thus we can partition  $V(\Pi')$  into two sets — the set  $V_1$  containing those variables which get redefined before use when  $\Pi'$  executes from  $s'$  and the set  $V_2$  containing the variables which have the same value in  $s'$  and  $s''$ . Validity requires that the union of these two sets is  $V(\Pi')$ . Note that the two sets need not be disjoint i.e. there could be variables which have the same value in  $s'$  and  $s''$  and are also defined before use when  $\Pi'$  executes from  $s'$ .

Thus to use Theorem 3.2 to determine the control points of  $\Pi$  at which the on-line change can be safely installed without violating validity (i.e. to specify the possible times of change), we need to find the sets  $V_1$  and  $V_2$  for each control point  $C$  of  $\Pi$  and then select those control points for which  $V_1 \cup V_2 = V(\Pi')$ . However, finding the sets  $V_1$  and  $V_2$  is not decidable either. This is because it is undecidable as to whether a particular statement of a program is reachable or not for some input. Thus it is not even decidable to say whether there exists a reachable state  $s''$  of  $\Pi'$  with  $s''(PC) = C'$ . “Safe” estimates for the sets  $V_1$  and  $V_2$  can however be obtained by data flow analysis. These estimates can be used to find *some* control points at which the conditions of Theorem 3.2 are satisfied.

---

<sup>2</sup>Strictly speaking, this is true only if there are no variables of  $\Pi'$  which are never used when  $\Pi'$  is executed from  $s'$ . But clearly we need not worry about such variables since they can not affect the behavior of the program in any way.

A safe estimate for the set  $V_1$  can be obtained by using the standard live variable analysis done in compilers for the purpose of code optimization [ASU86]. This analysis gives the set of variables which *may* be used before definition when execution starts from a particular point in a program. Subtracting this set from the set of all the variables of the program gives the set of variables which will decidedly be defined before any use. In general, by this method, the set  $V_1$  of variables that we get is a subset of the actual set of variables that are redefined before use. We will give an algorithm to compute a safe estimate for the set  $V_2$  in the next section.

Till now we have not made any assumption about the nature of the mapping  $\mathcal{C}$ . In practice,  $\Pi$  and  $\Pi'$  are usually related programs and one would want  $\Pi'$  to start executing “where  $\Pi$  left off”. Thus the mapping  $\mathcal{C}$  will usually map a control point of  $\Pi$  to an “equivalent” control point of  $\Pi'$ . For this reason, and because it is very difficult to compute the set  $V_2$  for an arbitrary  $\mathcal{C}$ , we assume that for a control point  $C$  of  $\Pi$ ,  $\mathcal{C}(C)$  is the “equivalent” control point of  $C$  in  $\Pi'$  if one exists and is undefined otherwise. We now formalize the notion of a control point and specify more precisely what is meant by the “equivalence” of two control points of  $\Pi$  and  $\Pi'$  respectively.

For this we need to characterize the differences between the two programs  $\Pi$  and  $\Pi'$ . This characterization is done by means of two sets  $D_{\Pi,\Pi'}$  and  $D_{\Pi',\Pi}$  of statements of  $\Pi$  and  $\Pi'$  respectively.  $D_{\Pi,\Pi'}$  contains the statements of  $\Pi$  which are not in  $\Pi'$  and  $D_{\Pi',\Pi}$  contains the statements of  $\Pi'$  which are not in  $\Pi$ . In order to compute these sets we need the notion of “equality” of expressions and statements of  $\Pi$  and  $\Pi'$  respectively. We assume the existence of a given variable mapping  $\mathcal{V}$ .

We consider two expressions  $E$  and  $E'$ , appearing in  $\Pi$  and  $\Pi'$  respectively as *different* if and only if one of the following is true.

1. The main operators, if any, of  $E$  and  $E'$  are different.
2. One of  $E$  and  $E'$  is a variable and the other is a constant.
3.  $E$  and  $E'$  are different constants.
4.  $E$  and  $E'$  are both variables and either  $\mathcal{V}(E)$  is not defined or  $\mathcal{V}(E) \neq E'$ .

5. The main operators of  $E$  and  $E'$  are the same but one of the operands of  $E$  is different from the corresponding operand of  $E'$  according to this definition.

Similarly, we consider statements  $S$  and  $S'$  of  $\Pi$  and  $\Pi'$  as *different* if and only if one of the following is true.

1.  $S$  and  $S'$  are of different kinds (for example, one is a while-do statement and the other is an assignment statement).
2.  $S$  is  $x := E$ ,  $S'$  is  $x' := E'$  and  $\mathcal{V}(x)$  is not defined or  $\mathcal{V}(x) \neq x'$  or  $E$  and  $E'$  are different.
3.  $S$  and  $S'$  are both if-then, if-then-else or while-do type of statements and the conditional expressions of  $S$  and  $S'$  are different.

The desired property of the sets  $D_{\Pi, \Pi'}$  and  $D_{\Pi', \Pi}$  is that  $\Pi$  can be made the same as  $\Pi'$  by deleting the statements in  $D_{\Pi, \Pi'}$  from  $\Pi$ , applying  $\mathcal{V}$  to the resulting program and then adding the statements in  $D_{\Pi', \Pi}$  to it. Clearly, this allows all the statements of  $\Pi$  to be in  $D_{\Pi, \Pi'}$  even if the two programs are nearly identical. This will result in our getting much more conservative estimates from the data flow analysis. Therefore, we want the sets  $D_{\Pi, \Pi'}$  and  $D_{\Pi', \Pi}$  to be as small as possible. In Appendix A, we give a method to compute  $D_{\Pi, \Pi'}$  and  $D_{\Pi', \Pi}$  which, except in rare circumstances, finds the smallest possible sets of statements. For each statement  $S$  of  $\Pi$  not in  $D_{\Pi, \Pi'}$ , the algorithm also gives “corresponding( $S$ )” which specifies the statement of  $\Pi'$  which is not different from  $S$ . The “corresponding” function is one-to-one and is such that if it is defined for  $S$  then  $S$  and corresponding( $S$ ) are not in  $D_{\Pi, \Pi'}$  and  $D_{\Pi', \Pi}$  respectively and if it is defined for  $S_1$  and  $S_2$  and  $S_1$  precedes  $S_2$  in  $\Pi$  then corresponding( $S_1$ ) textually precedes corresponding( $S_2$ ) in  $\Pi'$ .

We can now formalize the notion of control points and of “equivalence” between two control points of  $\Pi$  and  $\Pi'$  respectively.

**Definition 3.1** A *control point*  $C$  of  $\Pi$  is a pair  $(S_1, S_2)$  of consecutive statements of  $\Pi$ . At most one of  $S_1$  and  $S_2$  can be  $\phi$  —  $S_1$  if  $S_2$  is the first statement of a block and  $S_2$  if  $S_1$  is the last statement of a block.

If for a state  $s$ ,  $s(\text{PC}) = (S_1, S_2)$ , it means that the execution has reached after  $S_1$  but before  $S_2$ , in the state  $s$ .

**Definition 3.2** A control point  $(S_1, S_2)$  of  $\Pi$  has an *equivalent* in  $\Pi'$  if and only if either  $S_1 \notin D_{\Pi, \Pi'}$  and  $\text{corresponding}(S_1) \notin D_{\Pi', \Pi}$  or  $S_2 \notin D_{\Pi, \Pi'}$  and  $\text{corresponding}(S_2) \notin D_{\Pi', \Pi}$ . Also none of the super-statements containing  $S_1$  (or  $S_2$ <sup>3</sup>) as a sub-statement are in  $D_{\Pi, \Pi'}$  and none of the super-statements containing  $\text{corresponding}(S_1)$  or  $\text{corresponding}(S_2)$  (whichever is defined) as a sub-statement are in  $D_{\Pi', \Pi}$ .

**Definition 3.3** If the control point  $(S_1, S_2)$  satisfies the conditions of Definition 3.2 then its equivalent in  $\Pi'$  is the control point in  $\Pi'$  just after the statement  $S'_1 = \text{corresponding}(S_1)$  if  $S_1 \notin D_{\Pi, \Pi'}$  and the control point in  $\Pi'$  just before the statement  $S'_2 = \text{corresponding}(S_2)$  otherwise.

We now assume that the mapping  $\mathcal{C}$  is such that it maps any control point  $C$  of  $\Pi$  to its equivalent control point in  $\Pi'$  if it exists and is undefined otherwise. The problem of applying Theorem 3.2 now reduces to finding the sets  $V_1$  and  $V_2$  for each pair of equivalent control points of  $\Pi$  and  $\Pi'$  respectively. The sets  $V_1$  can be found by live variable analysis as explained earlier. In the next section, we give an algorithm to find safe estimates for the sets  $V_2$ .

### 3.5 Algorithm *Change*

Now we present the algorithm *Change* using which we can estimate the set  $V_2$  for each control point  $C$  of  $\Pi$  with an equivalent in  $\Pi'$ . In other words, we can determine the set of variables which would have had the same values had  $\Pi'$  been executed since the beginning instead of  $\Pi$ . Given  $\Pi$ , for each control point  $C$  of  $\Pi$  for which  $\mathcal{C}(C)$  is defined (i.e.  $C$  has an equivalent in  $\Pi'$ ) the algorithm gives a set  $\text{changed}(C)$  of variables of  $\Pi$ . This set is the set of variables  $x$  such that  $x$  is either defined by a statement in  $D_{\Pi, \Pi'}$  or by a statement which uses another such variable and this definition of  $x$  “reaches”  $C$ .

---

<sup>3</sup>Since  $S_1$  and  $S_2$  are consecutive statements, they are the sub-statements of the same statement.

The algorithm is inductive in nature. Given the *changed* set before a statement, it computes the *changed* set after the statement depending on the type of the statement and whether it is in  $D_{\Pi, \Pi'}$  or not. This set is computed by adding a *gen* set to the old set and removing a *kill* set from it. The *gen* and the *kill* sets are defined as follows.

**Definition 3.4**

$$\text{gen}(S) = \begin{cases} \{x \mid S \text{ defines } x \text{ and uses a variable in } \text{changed}(S).\} \\ \quad \text{if } S \notin D_{\Pi, \Pi'} \\ \{x \mid S \text{ defines } x\} & \text{otherwise} \end{cases}$$

Here  $\text{changed}(S)$  denotes the *changed* set for the control point just before the statement  $S$ .

**Definition 3.5**

$$\text{kill}(S) = \begin{cases} \{x \mid S \text{ defines } x \text{ and does not use any variable in } \text{changed}(S).\} \\ \quad \text{if } S \notin D_{\Pi, \Pi'} \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 3.1 gives the formal recursive algorithm *Change*. Given a sequence of statements  $\bar{S}$  and an initial set of variables that might have changed before the execution of the sequence (the *in* set), it computes  $\text{changed}(C)$  for all control points  $C$  in  $\bar{S}$  just before statements which are not in  $D_{\Pi, \Pi'}$ . and returns the set of variables that might have changed after the execution of the sequence. When called as  $\text{Change}(\Pi, \emptyset)$ , it computes  $\text{changed}(C)$  for all control points  $C$  of  $\Pi$  just before statements not in  $D_{\Pi, \Pi'}$ .

The above algorithm is invoked as  $\text{Change}(\Pi, \emptyset)$  after initially computing  $D_{\Pi, \Pi'}$  by some method (the method given in Appendix A for example). Essentially, the algorithm is a syntax directed forward direction data-flow analysis [ASU86]. It scans the statements of  $\Pi$  one by one. For a simple statement or a statement in  $D_{\Pi, \Pi'}$ , the new in-set is computed by adding  $\text{gen}(S)$  to and removing  $\text{kill}(S)$  from the old one. For an if-then statement, the algorithm is recursively called for the sequence of statements which constitutes the “then-part” of the statement. Now there are two possible paths to the statement following the if-then statement, either by executing

**Algorithm** *Change*( $\bar{S}$ , in-set)  
**begin**  
 $C \leftarrow$  control point before  $\bar{S}$ ;  
**while** there is a statement after  $C$  **do begin**  
 $S \leftarrow$  Statement after  $C$ ;  
**if**  $S \in D_{\Pi, \Pi'}$  or  $S$  is a simple statement **then begin**  
**if**  $S$  is a while statement **then**  
in-set  $\leftarrow$  in-set  $\cup$  gen( $S$ );  
changed( $C$ )  $\leftarrow$  in-set;  
 $C \leftarrow$  control point after  $S$ ;  
in-set  $\leftarrow$  gen( $S$ )  $\cup$  (in-set  $\perp$  kill( $S$ ));  
**end else if** in-set  $\cap$  cond-vars( $S$ )  $\neq \emptyset$  **then**  
 $D_{\Pi, \Pi'} \leftarrow D_{\Pi, \Pi'} \cup \{S\}$ ;  
**else if**  $S$  is an if-then statement **then begin**  
changed( $C$ )  $\leftarrow$  in-set;  
 $C \leftarrow$  control point after  $S$ ;  
in-set  $\leftarrow$  in-set  $\cup$  change(then-part of  $S$ , in-set);  
**end else if**  $S$  is an if-then-else statement **then begin**  
changed( $C$ )  $\leftarrow$  in-set;  
 $C \leftarrow$  control point after  $S$ ;  
in-set  $\leftarrow$  change(then-part of  $S$ , in-set)  $\cup$   
change(else-part of  $S$ , in-set);  
**end else begin** /\*  $S$  is a while statement \*/  
**repeat**  
old-in  $\leftarrow$  in-set;  
in-set  $\leftarrow$  in-set  $\cup$  change(loop of  $S$ , in-set);  
**until** in-set = old-in **or** in-set  $\cap$  cond-vars( $S$ )  $\neq \emptyset$   
**if** in-set  $\cap$  cond-vars( $S$ )  $\neq \emptyset$  **then**  
 $D_{\Pi, \Pi'} \leftarrow D_{\Pi, \Pi'} \cup \{S\}$ ;  
**else begin**  
changed( $C$ )  $\leftarrow$  in-set;  
 $C \leftarrow$  control point after  $S$ ;  
**end**  
**end**  
**end**  
**return** in-set;  
**end.**

Figure 3.1: Algorithm *Change*

the then-part or by bypassing it. Correspondingly, to safely estimate the in-set after the if-then statement, the new in-set is computed by taking the union of the old one and the out set returned by the recursive call. The case of the if-then-else statement is handled similarly. In the case of a while statement, there are infinitely many possible paths to the statement following it corresponding to the number of times the loop is executed. Furthermore, the control point just before the while statement can also be reached through an infinite number of paths. Therefore, the algorithm iterates till the in-set stabilizes. Stabilization is guaranteed in a finite number of iterations, since, as can be easily seen, the set always grows. Another point to note about the algorithm is that it changes the  $D_{\Pi, \Pi'}$  set. Whenever a compound statement is encountered such that some of the variables occurring in the condition of the statement (the  $\text{condvars}(S)$  set) are also in the in-set, the statement is added to  $D_{\Pi, \Pi'}$  and treated as such. This is because a control point before a sub-statement of such a statement may not have a “corresponding” control point in  $\Pi'$  because had  $\Pi'$  been executed instead of  $\Pi$ , the condition might have evaluated to a different value and the control may not have reached the corresponding sub-statement in  $\Pi'$  at all.

To find  $\text{changed}(C')$  for a control point  $C'$  of  $\Pi'$ , the same algorithm is used but with  $\Pi'$  and  $D_{\Pi', \Pi}$  instead of  $\Pi$  and  $D_{\Pi, \Pi'}$  respectively.

It can be shown that for any control point  $C$  of  $\Pi$  which has an equivalent  $C'$  in  $\Pi'$ , the set  $V(\Pi') \perp (V_C \cup V_{C'})$  is a safe approximation for (i.e. a subset of) the set  $V_2$  where  $V_C = \mathcal{V}(\text{changed}(C))$ ,  $V_{C'} = \text{changed}(C')$  and for a set  $V$ ,  $\mathcal{V}(V)$  represents  $\{\mathcal{V}(x) | x \in V \text{ and } \mathcal{V}(x) \text{ is defined}\}$ . That is, for any reachable state  $s$  of  $\Pi$  with  $s(\text{PC}) = C$ , there exists a reachable state  $s'$  of  $\Pi'$  with  $s'(\text{PC}) = C'$  such that  $s'(x) = s(\mathcal{V}^{-1}(x))$  for all  $x \in V(\Pi') \perp (V_C \cup V_{C'})$ . The proof of this claim is given in Appendix B.

### 3.6 An Example

We now give a simple example to illustrate the approach discussed above. We consider a simple program which reads a number and prints its factorial. This is repeated in an endless cycle. Two versions of the program are shown in Figure 3.2.

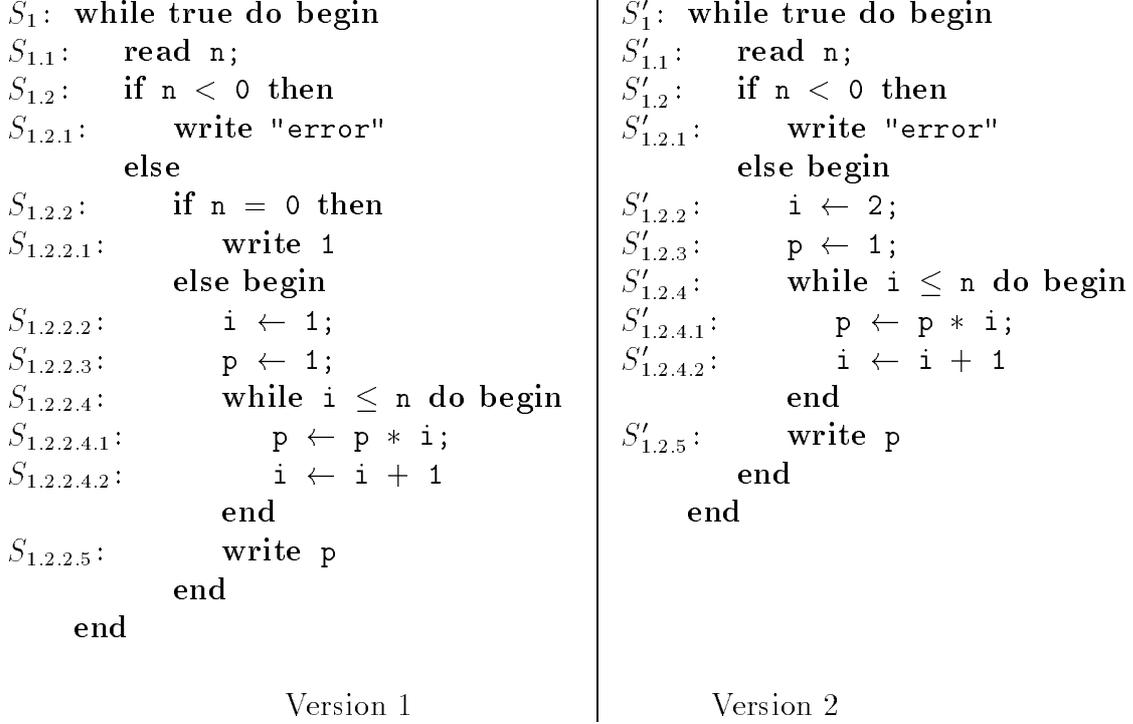


Figure 3.2: The two versions of the factorial program

The statements have been labelled for ease of future reference. It can be seen that  $V(\Pi) = V(\Pi') = \{i, n, p\}$ . We take  $\mathcal{V}$  as the identity mapping. Using the procedure given in Appendix A,  $D_{\Pi, \Pi'}$  and  $D_{\Pi', \Pi}$  are computed as the following sets.

$$\begin{aligned}
 D_{\Pi, \Pi'} &= \{S_{1.2.2}\} \\
 D_{\Pi', \Pi} &= \{S'_{1.2.2}, S'_{1.2.3}, S'_{1.2.4}, S'_{1.2.5}\}
 \end{aligned}$$

Further, for statements  $S_m$  of  $\Pi$  not in  $D_{\Pi, \Pi'}$ , we get  $\text{corresponding}(S_m) = S'_m$ .

The algorithm *Change* is now run. The values of the changed sets for various control points along with the sets  $V_1$  and  $V_2$  are shown in Table 3.1. The sets  $V_C$  and  $V_{C'}$  represent  $\mathcal{V}(\text{changed}(C))$  and  $\text{changed}(C')$  respectively.  $V_1$  is the set of variables which will be defined before use when execution of  $\Pi'$  starts from  $C'$  (this set can be determined by live variable analysis) and  $V_2$  is  $V(\Pi') \perp (V_C \cup V_{C'})$ .

It can be seen that for all the control points in  $\Pi$  which have equivalents in  $\Pi'$ ,

$C$	$C'$	$V_1$	$V_C$	$V_{C'}$	$V_2$	$V_1 \cup V_2$
$(\phi, S_1)$	$(\phi, S'_1)$	$\{i, n, p\}$	$\{i, p\}$	$\{i, p\}$	$\{n\}$	$\{i, n, p\}$
$(\phi, S_{1.1})$	$(\phi, S'_{1.1})$	$\{i, n, p\}$	$\{i, p\}$	$\{i, p\}$	$\{n\}$	$\{i, n, p\}$
$(S_{1.1}, S_{1.2})$	$(S'_{1.1}, S'_{1.2})$	$\{i, p\}$	$\{i, p\}$	$\{i, p\}$	$\{n\}$	$\{i, n, p\}$
$(\phi, S_{1.2.1})$	$(\phi, S'_{1.2.1})$	$\{i, p\}$	$\{i, p\}$	$\{i, p\}$	$\{n\}$	$\{i, n, p\}$
$(S_{1.2.1}, \phi)$	$(S'_{1.2.1}, \phi)$	$\{i, p\}$	$\{i, p\}$	$\{i, p\}$	$\{n\}$	$\{i, n, p\}$
$(S_{1.2}, \phi)$	$(S'_{1.2}, \phi)$	$\{i, n, p\}$	$\{i, p\}$	$\{i, p\}$	$\{n\}$	$\{i, n, p\}$

Table 3.1: Ensuring validity of the on-line change to the factorial program

$(V_1 \cup V_2) = V(\Pi')$ . Thus validity is ensured if the on-line change is installed at any of these control points.

### 3.7 A More General Program Model

We now extend our model to allow functions and procedures in the language. In this model, the programs can have functions and procedures with input and input/output type of parameters. For the sake of simplicity, we assume that the latter are implemented by means of a copy-in and copy-out mechanism. Global variables are also allowed. Each program has a function called `main` where the execution normally starts. The semantics of the language are the same as for any conventional procedural language.

Any reasonable size program consists of a number of functions and procedures. In most systems, for performing an on-line change, frequently a function<sup>4</sup> or a set of functions is changed. In other words, in large systems, frequently the aim is to replace some existing function by a new function. If we try to apply the conditions discussed in the previous sections to this case, we will have to do inter-procedural data flow analysis which, as is well known, is not only computationally very expensive but also gives highly conservative estimates. We will therefore try to obtain simpler conditions for validity by treating functions as the units of change.

With a function as the unit of change, the whole function is considered to be

---

<sup>4</sup>We will henceforth use the word function for both procedures and functions.

changed even if only a small part of it is modified. In other words, we do not view the internals of a function, regardless of the number of statements changed in it. This implies that we can not control the timing of an on-line change by specifying a set of control points at which the change is acceptable, as we did earlier. We can only specify that the change be done when the control point is within some function. That is, we can not specify *exactly* the control point at which the change should be done but only specify the function it should be in. However, the approach we take is to specify the functions the control should *not* be in at the time of change. We intend to develop conditions which specify that the control is not within some functions by stating that the functions are not on the stack. This is a general method which can handle the call hierarchy (where the control can be inside many functions at a given time.)

For simplicity, we assume that functions are only changed and not added or deleted. The case of addition and deletion can very simply be handled by considering the added (deleted) functions to be present in the old (new) version also but without being called from any other function. Thus, in fact we need not even consider the added or deleted functions as changed.

It is clear that at the very least, the changed functions should not be on the stack at the time of change, for otherwise we would have to look inside the functions to map the control points to some corresponding value for the new versions of the functions. For unchanged functions however, the control point of the new version corresponding to the given control point in the old version can be determined without “looking inside” the function just by mapping the control point to the same offset in the new function. It can be easily seen that only having the changed functions off the stack at the time of change does not always lead to validity. Therefore our goal for validity is

*To find the set of functions  $F$  (which always includes the set of changed functions) such that validity is ensured if none of the functions in  $F$  are on the stack at the time of change.*

### 3.7.1 Model of State

For this model, we must use a somewhat more elaborate model of state than the previous one since we must now capture the stack as well. We therefore define the state of a process executing some program as follows.

**Definition 3.6** A state is a function which maps triplets of the form

$$(\langle \text{function name} \rangle, \langle \text{position on stack} \rangle, \langle \text{variable name or PC} \rangle)$$

and

$$(-, -, \langle \text{variable name} \rangle)$$

to values in the appropriate domains.

The state is a mapping which gives values to different variables (or PC) in a context (the first two components of a triplet). Triplets are chosen so that a variable can be uniquely specified. Triplets of the form  $(-, -, x)$  describe global variables and those of the form  $(f, p, x)$  describe local variables and function arguments of the function  $f$  (the  $p$ th stack frame on the stack). Triplets of the form  $(f, p, PC)$  give the current program counter and the various return values from the stack. Since the triplets also include the position on the stack, it is possible for a function to have more than one stack frame in the stack and thus recursive calls are also allowed.

As an example, if functions `main`, `f` and `g`, with local variables `x`, `y` and `z` respectively, of a program are on the run-time stack at a point in time, and `u` and `v` are the global variables of the program, then the corresponding state maps only the triplets  $(\text{main}, 1, x)$ ,  $(\text{main}, 1, PC)$ ,  $(f, 2, y)$ ,  $(f, 2, PC)$ ,  $(g, 3, z)$ ,  $(g, 3, PC)$ ,  $(-, -, u)$  and  $(-, -, v)$  to legal values and all other possible triplets have undefined values. The value of the triplet  $(g, 3, PC)$  is the value of the program counter in the state and the value of  $(f, 2, PC)$  is the return address from `g` to `f`.

In the initial state  $s_{\Pi_0}$  of a program  $\Pi$ , the value of the triplet  $(\text{main}, 1, PC)$  is the first control point in the function `main`, for a global variable  $x$  the value of  $(-, -, x)$  is  $\perp$ , for a local variable  $y$  of `main` the value of  $(\text{main}, 1, y)$  is  $\perp$  and no other triplets have defined values. As before, we assume that using a variable with the value  $\perp$  has unpredictable results. If  $s_1$  is a reachable state of a program  $\Pi$  and  $s_2$  is the same

as  $s_1$  except for the values of some variables which have the value  $\perp$  in  $s_1$  then, as a convention, we consider  $s_2$  also a reachable state of  $\Pi$ .

We consider only a simple class of state mappings which allow only mapping of global variable values. Specifically, any mapping  $\mathcal{S}$  in this class is such that for any state  $s$ ,  $\mathcal{S}(s)$  is defined if and only if no changed functions are on the stack in  $s$ , all triplets representing local variables and PC values have the same values in  $s$  and  $\mathcal{S}(s)$  and the value of any global variable in  $\mathcal{S}(s)$  is a function only of the values of some global variables in  $s$ .

Based on the mapping  $\mathcal{S}$ , we define the following sets of variables.

The set of *mapped variables* is the set of all global variables  $x$  of  $\Pi'$  such that either  $x$  is not a global variable of  $\Pi$  or for some state  $s$ ,  $(\mathcal{S}(s))((-,-,x)) \neq s((-,-,x))$ . The set of the remaining global variables of  $\Pi'$  is called the set of *unmapped variables*. Thus unmapped variables are those which always have the same value in the mapped state and the original state.

The set of *control variables* is the set of global variables  $x$  of  $\Pi$  such that the value of some mapped variable  $y$  of  $\Pi'$  in  $\mathcal{S}(s)$  is a function of the value of  $x$  in  $s$ . Thus, if in two states  $s_1$  and  $s_2$ , for all control variables  $x$ ,  $s_1((-,-,x)) = s_2((-,-,x))$  then for all mapped variables  $y$  of  $\Pi'$ ,  $(\mathcal{S}(s_1))((-,-,y)) = (\mathcal{S}(s_2))((-,-,y))$ .

In practical situations, since there is likely to be very little difference in the two versions of the programs, the sets of mapped and control variables are likely to be very small.

### 3.7.2 Validity of Change

There are two possible ways of ensuring validity in this model. The first one is to ensure that the change is in a state such that the mapped state is a reachable state for the new version, by putting restrictions on the type of change.<sup>5</sup> This solution, however, restricts the types of changes permitted. The other solution, which is more general, ensures that the process reaches a reachable state of the new version some time after the change. We discuss the first solution first.

---

<sup>5</sup>This possibility was in the simpler model also but was subsumed by the conditions of Theorem 3.2 which as shown later are not practicable in this case.

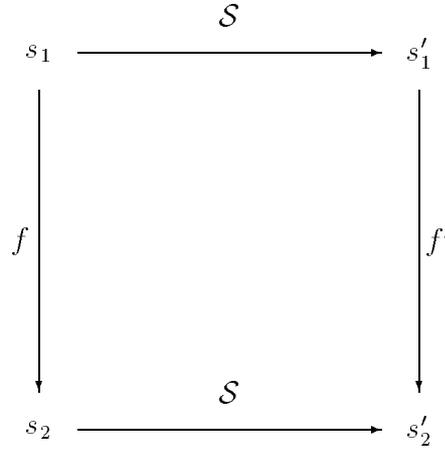


Figure 3.3: Diagrammatic representation of functional enhancement

### 3.7.3 Change in a Reachable State of $\Pi'$

Validity can be ensured easily if restrictions are imposed such that the change is always effected in a state  $s$  such that  $\mathcal{S}(s)$  is a reachable state for  $\Pi'$ . This will mostly be true for changes in which the changed function is closely related in its behavior to its new version. One possibility is that the new version of the changed function is a “functional enhancement” of the old version. We formally define the notion of functional enhancement below.

**Definition 3.7** The function  $f'$  is said to be a *functional enhancement* of  $f$  with respect to the mapping  $\mathcal{S}$  if the following holds. Let  $s_1$  be a reachable state of  $\Pi$  for which  $\mathcal{S}$  is defined and the statement to be executed next in  $s_1$  is a call to  $f$ . If in a process executing  $\Pi$ ,  $f$  is called in state  $s_1$  and the resultant state just after it returns is  $s_2$ , then if  $f'$  is called in a process executing  $\Pi'$  in state  $\mathcal{S}(s_1)$ , there exist some inputs for which  $f'$  will return and the state just after its return will be  $\mathcal{S}(s_2)$ .

Intuitively, the change to a function is an enhancement if the operations of executing  $f$  or  $f'$ , and applying  $\mathcal{S}$  are commutative, as shown Figure 3.3. Thus the new version of a functionally enhanced function computes the same or a “larger” function i.e. it is defined for more input states or that for the same state it has a

larger set of resultant output states. In practice, this will happen if, for example, the specifications of the changed function do not change and the change is made for reasons of efficiency improvement or for changing the output format etc. Another possible situation in which the change is a functional enhancement is the following. Suppose, the statement  $S_1$  of a function  $f$  is replaced by the statement sequence “**read  $x$ ; if  $\text{cond}(x)$  then  $S_1$  else  $S_2$ ;**” in the new version, where  $x$  is a new variable. Clearly, the new version of the program is offering more functionality to the user and the change to the function  $f$  is a functional enhancement. The change could also, for example, be adding a new case to a case statement. Such a situation might arise if, for example, the changed function presents a menu to the user and the new version offers a larger menu. An exhaustive list of the possibilities can obviously not be made.

Note that in the context of on-line change, we are concerned with the functionality of the actual function code and not its stated functionality (or specification) which may not be satisfied by the actual code if it contains bugs. Thus if the newer version of a function fixes a bug of the older function, it need not be a functional enhancement even though the intended functionality of the two versions is the same. We can now state the following result.

**Theorem 3.3** *For any reachable state  $s$  of  $\Pi$  for which  $\mathcal{S}$  is defined,  $\mathcal{S}(s)$  is a reachable state of  $\Pi'$  if the new version  $f'$  of each changed function  $f$  is a functional enhancement of  $f$  with respect to  $\mathcal{S}$ , no mapped variables are used or defined by any unchanged functions and no control variables are defined by any unchanged functions.*

**Proof:** The result is proved by induction on the sequence of states (starting with  $s_{\Pi_0}$ ) through which  $s$  has been reached. In the base case,  $s$  is  $s_{\Pi_0}$ . In this case,  $\mathcal{S}(s)$  is the same as  $s_{\Pi'_0}$  except that the global variables may have values different from  $\perp$ . Thus  $\mathcal{S}(s)$  is a reachable state of  $\Pi'$  by convention.

As the induction hypothesis we assume the result to be true of a reachable state  $s_1$  of  $\Pi$  and in the induction step, show the result to be true of the first state  $s_2$  reached from  $s_1$  such that  $\mathcal{S}(s_2)$  is defined. We consider the various ways in which  $s_2$  could have been reached from  $s_1$ . This depends on the statement to be executed

next in  $s_1$ . This statement is the same in both  $s_1$  and  $\mathcal{S}(s_1)$  because  $\mathcal{S}$  is defined only when no changed functions are on the stack and it does not alter the PC triplet values. Further, this statement can not use or define any mapped variables and can not define any control variables.

If in  $s_1$  the statement to be executed next is a call to a changed function  $f$  then  $s_2$  is the state after the return of  $f$  and  $\mathcal{S}(s_2)$  is a reachable state of  $\Pi'$  because when  $\Pi'$  executes from  $\mathcal{S}(s_1)$  (a reachable state of  $\Pi'$  by the induction hypothesis) the state after the return of  $f$  can be  $\mathcal{S}(s_2)$  since the new version of  $f$  is a functional enhancement of the old one.

If the statement to be executed next in  $s_1$  is not a call to a changed function then  $s_2$  is the state immediately following  $s_1$ . Let  $s'_2$  be the state reached from  $\mathcal{S}(s_1)$  after the execution of this statement in  $\Pi'$ . We show that  $s'_2 = \mathcal{S}(s_2)$  which implies that  $\mathcal{S}(s_2)$  is a reachable state of  $\Pi'$  since  $\mathcal{S}(s_1)$  is assumed to be a reachable state of  $\Pi'$ .

From the properties of  $\mathcal{S}$  it follows that all triplets representing local variables, PC values and unmapped variables have the same values in  $s_1$  and  $\mathcal{S}(s_1)$ . The same property holds for  $s_2$  and  $\mathcal{S}(s_2)$  as well. Since the statement between  $s_1$  and  $s_2$  does not use any mapped variables, the same transformation is performed on the rest of the state in going from  $s_1$  to  $s_2$  as in going from  $\mathcal{S}(s_1)$  to  $s'_2$ <sup>6</sup>. Thus the values of all triplets representing local variables, PC values and unmapped variables are the same in  $s_2$  and  $s'_2$  and hence also in  $\mathcal{S}(s_2)$  and  $s'_2$ .

Now all control variables must have the same values in  $s_1$  and  $s_2$  because the statement between them can not define any control variable. It follows that all mapped variables must have the same values in  $\mathcal{S}(s_1)$  and  $\mathcal{S}(s_2)$ . Also all mapped variables have the same values in  $\mathcal{S}(s_1)$  and  $s'_2$  since the statement between  $\mathcal{S}(s_1)$  and  $s'_2$  can not define any mapped variable. Thus all mapped variables have the same value in  $\mathcal{S}(s_2)$  and  $s'_2$ . It follows that  $s'_2 = \mathcal{S}(s_2)$ . ■

The above theorem states that in the case of functional enhancement, a change effected in any state in which the state mapping is defined (in other words, the changed functions are not on the stack) is valid. The restriction that no mapped

---

<sup>6</sup>If the statement between  $s_1$  and  $s_2$  is an input statement, there are several possibilities for  $s'_2$ . In this case, we choose  $s'_2$  as the state which would have been obtained from  $\mathcal{S}(s_1)$  if the same value is read as in going from  $s_1$  to  $s_2$ .

variables should be used or defined by any unchanged functions is quite natural and to be expected since the computation of these variables is different in the two versions. The requirement that control variables should also not be defined by unchanged functions is however a bit surprising but it can be shown that without this restriction the result does not always hold.

Note that the functionality of a function may be affected even if the function is not changed itself but changes are made to one or more functions which it calls. Thus the definition of functional enhancement can be meaningfully applied even to an unchanged function. For Theorem 3.3 to hold, however, we only require the changed functions to be functionally enhanced. In fact it can be easily verified that all the unchanged functions are also functionally enhanced if all the changed ones are. We now give a generalization of Theorem 3.3 which does not require all the changed functions to be functionally enhanced.

**Theorem 3.4** *For any reachable state  $s$  of  $\Pi$ ,  $\mathcal{S}(s)$  is a reachable state of  $\Pi'$  if there is a set of functions  $F$  such that the new versions of all functions in  $F$  are functional enhancements of their respective older versions and in the call graph of  $\Pi'$ , each path from `main` to a changed function passes through a function in  $F$ , all functions in  $F$  are off the stack in  $s$ , no mapped variables are used or defined by any unchanged function not in  $F$  and no control variables are defined by any unchanged function not in  $F$ .*

**Proof:** Similar to the proof for Theorem 3.3 ■

Intuitively, a functionally enhanced function acts as a firewall by protecting the state from the effect of execution of a changed function i.e., it does not allow the effect on state due to the execution of the older version instead of the newer one to propagate to its (the functionally enhanced function's) caller. For validity, the effects of execution of changed functions should not reach `main`, therefore the theorem requires a functionally enhanced function to be on each path in the call graph from `main` to any changed function.

As an example of the use of this theorem, consider a function `sort_print`, which given a list of records, sorts it (into a local list) and prints it. For sorting it uses the function `sort` which calls function `compare` for performing comparisons. Assume

that `sort` is only called by `sort_print` and `compare` is only called by `sort`. Now suppose, the function `compare` is to be replaced by a version which perhaps compares some other field of the records. In this case, the new version of `compare` is not a functional enhancement of its old one but the function `sort_print` in the new version of the program is a functional enhancement of the same function in the old version and every path from `main` to `compare` passes through `sort_print`. Thus by Theorem 3.4, the on-line change is guaranteed to be valid if done at any time when `sort_print` is off the stack. This can be easily verified intuitively also.

To use Theorem 3.4, one would normally find the smallest set  $F$  which satisfies the requirements of the theorem, so that the restrictions on the time of change are as lax as possible. However even the smallest such set  $F$  may contain some functions very near to `main` in the call hierarchy of  $\Pi$  and requiring all the functions in  $F$  to be off the stack at the time of change may be a rather severe restriction on the timing of change and may be satisfied only a large time after the user has expressed his wish to install the change or even never at all. In such situations, other means of achieving validity must be considered.

The conditions of Theorem 3.4 are only sufficient and not necessary. It follows that in certain cases, even though the requirements of this theorem are not met, the user may be able to determine that the mapped state will be a reachable state of  $\Pi'$  if certain functions are kept off the stack.

We do not give an algorithm for checking whether the new version of a function is a functional enhancement of the old one since it would involve inter-procedural data-flow analysis which we wish to avoid. Moreover, such an algorithm would only be able to detect only the simpler cases of enhancement and only for a much simpler class of state mappings. For example, if the new version of a function has the same functionality as the old one but is implemented using a totally different algorithm, it is not, in general, possible to automatically detect that this change is a functional enhancement. The programmer, on the other hand, is likely to have this information since he has semantic knowledge of the functionality of the two versions of the function. We therefore leave it to the programmer to determine this relationship based on his semantic knowledge of the programs.

### 3.7.4 Change in an Unreachable State of $\Pi'$

The above approach obviously restricts the types of changes that can be allowed. To take care of the more general case, we try to relate it with the conditions given for the simpler model. Since having or not having functions on the stack at the time of change is the only control we now have on the timing of change, for a function not required to be off the stack at the time of change, it is obvious that the change must be valid for *all* the control points within the function. In terms of conditions of Theorem 3.2, it means that for all the control points, it must be the case that the variables affected by the change are defined before use. This implies that the values computed by the changed function and affected by the change are not used at all! Since this is not reasonable to expect, the earlier sufficient conditions become overly restrictive in this case.

The proposed solution is to require that all the variables affected by the change and the variables that they subsequently affect are “ultimately” redefined. To formalize this, we define a set  $V_c$  of variables for each state of the process after the change. Informally,  $V_c$  is the set of variables which have so far been affected by the change. Our condition for validity then requires that  $V_c$  ultimately becomes empty. This condition is most likely to be true in continuously running programs which process more or less independent operations and the processing of these operations is mostly independent of the persistent state (i.e. variables whose values are used across operations).

We formalize this condition first for the simple program model (without functions).

**Condition 3.1** Let  $s' = \mathcal{S}(s)$  and  $s'(PC) = C'$ . There exists a reachable state  $s''$  of  $\Pi'$  with  $s''(PC) = C'$  such that  $V_c$  eventually becomes  $\emptyset$  for all possible inputs where  $V_c$  is defined precisely by specifying how it is computed as follows.

Initially (just after the change)

$$V_c = V(\Pi') \perp \{x | s''(x) = s'(x)\}.$$

Subsequently, after the execution of each statement, the new value of  $V_c$  is computed from the old value as follows. If a statement  $S$  defining  $x$  is executed then  $x$  is added

to  $V_c$  if  $S$  uses a variable in  $V_c$  and removed from  $V_c$  if it does not. If a compound statement  $S$  is executed such that some  $x$  appearing in its conditional expression is in  $V_c$ , then after the execution of any of its sub-statements, all the variables defined by the sub-statement are added to  $V_c$  and after the completion of the execution of the compound statement, all variables defined by any of its sub-statements are added to  $V_c$ . ■

It is easy to see that if Condition 3.1 is satisfied then if the execution of  $\Pi'$  starts from  $s'$ , it will eventually reach a reachable state of  $\Pi'$ . We can now state the following result for the simple program model (without functions.)

**Theorem 3.5** *In the simple program model, if Condition 3.1 is true for all reachable states  $s$  of  $\Pi$  and for the state mapping function  $\mathcal{S} = (\mathcal{C}, \mathcal{V})$ , then an on-line change from  $\Pi$  to  $\Pi'$  in any state  $s$  such that  $\mathcal{C}(s(PC))$  is defined using the state mapping  $\mathcal{S}$  is valid.*

**Proof:** Straight forward. ■

We now extend this condition to the more general program model with functions. In Condition 3.1, we required the state  $s''$  to be such that  $s''(PC) = s'(PC)$ . Correspondingly, we now require that the run-time stack be the same sequence of functions in the states  $s'$  and  $s''$  and that for any function  $F$  at position  $n$  on the stack,  $s'((F, n, PC)) = s''((F, n, PC))$ . The set  $V_c$  is also now a set of triplets (none of them with the third component as PC). The computation of  $V_c$  is correspondingly modified as follows. If a triplet becomes undefined as a result of executing a statement (a return statement, for example), it is removed from the set. It is easy to see that with these modifications, Theorem 3.5 holds in the extended program model as well.

It should be noted that the above condition being undecidable, it is not possible to give an exact algorithm to check it and a safe but approximate algorithm will again involve data-flow analysis. Therefore, we leave it to the programmer to ascertain using his knowledge of the program that this condition is satisfied.

There is a problem with the condition developed above. A static inspection of the program may show all the affected variables getting ultimately redefined. But at run-time, for a period after the change, the process is not in any reachable state

of the new version of the program. In this “transition” time, some functions may be called with their pre-conditions false. In such a situation, one can live with the function returning incorrect results (because the effect of these will be undone sooner or later) but it is also possible for the function to behave in a totally unexpected manner, for example, it may get stuck in an infinite loop or may cause an error which may cause the program to abnormally terminate (division by zero, for example).

One way to avoid this is to place some restrictions on the pre and post conditions of the changed functions. For example, consider a situation in which a function  $f$  is supposed to satisfy postcondition  $Q$ , given precondition  $P$  and the two versions  $f_1$  and  $f_2$  of the function satisfy postconditions  $Q_1$  and  $Q_2$  respectively, which are both stronger than  $Q$ , given preconditions  $P_1$  and  $P_2$  respectively, which are both weaker than  $P$ . It is clear that in this case after the change, no function will be called with its pre-condition false. The condition ensures that a function always returns during the transition phase if it does so under normal execution circumstances. Again this can be true of the calling function instead of the changed function itself in which case having the calling function off the stack at the time of change ensures validity. We note that these conditions again allow only a restricted type of changes in which the functionality of the program does not change and so do not solve the general problem.

In the more general case, the new version of the function to be changed, in this case, may be performing a distinctly different computation than the first version. There is no easy solution for this and we leave it to the programmer to determine, using his knowledge of the two versions of the program if these abnormal conditions will not occur or if it is possible to avoid them by having some more functions off the stack (other than the ones being changed) at the time of change. As an example consider a function  $f$ , which (besides other things) initializes a pseudo-random number generator by first calling a seed generating routine which generates a seed (from the current time of the day etc.) and then calls a random number initialization routine. Now suppose, the random number generator package is changed to a new version (perhaps to get a better probability distribution) and the new initialization routine requires a non-zero seed (because it divides something by the seed, perhaps).

Clearly the seed generating routine must be changed to guarantee a non-zero seed. But if the on-line change is made with  $f$  on the stack, the seed initialization routine may be given a zero seed resulting in a crash. The user with his knowledge of the program can determine, in this case that the function  $f$  must also be off the stack at the time of change.

# Chapter 4

## Implementation and Experience

In the last chapter, we saw how the conditions for validity can be simplified if we consider a function as the unit of change. Besides the simplicity of these conditions, treating a function as the unit of change is also more realistic for practical systems. In most practical situations, one would like to change some modules, in other words, replace some modules by their new versions. In this chapter, an approach for the implementation of on-line software version change and a prototype version change system based on this approach for supporting on-line changes with functions as the units of change are described. As seen in the last chapter, automatically determining which functions should be off the stack at the time of change in order to achieve validity is very difficult. For this reason, given the parameters of the change (the two program versions, the state mapping and the restrictions on the timing), the system only installs the on-line change and makes no attempt to automatically fix the timing of change. We describe the design and the implementation of the system and also discuss some experiments conducted to evaluate the system's performance and an experiment carried out to investigate certain engineering issues involved in on-line changes to programs

## 4.1 The Basic Approach and Design Principles

Many systems for on-line software version change have previously been described in the literature [Fab76, GIL78, Lee83, SF89a] (see Chapter 1 for more details). Most of these systems use indirect addressing tables to link the different modules. These tables are dynamically changed if a module has to be replaced by a new version. The changing of the table is done in a manner that it does not cause any inconsistencies. The major drawback of this approach is that the mechanisms used for dynamic updating are complicated and require new compilers and linkers to be written. A new approach is proposed herein which can be implemented in a simple and efficient manner.

The proposed approach to dynamic program version change is as follows. When a new version of the software is to be installed, a new process is created by the system with the new software. When the time is suitable for change (i.e. for ensuring validity), the system transfers the state of the old process (running the old software) to the new process (running the new software) and kills the old process. Thereafter, the new process runs.

The main design goal of the system was simplicity. This dictated that functions should be treated as the unit of change, so that the system does not have to “look inside” the functions. Further, keeping in mind the problems involved in automatically determining the time of change so as to ensure validity, as seen in the previous chapter, it was decided to make no attempt to do so but to have the user specify the timing of change in terms of the functions required off the stack at the time of change. Since the user is now responsible for ensuring the validity of change, he should be allowed to specify any arbitrary state mapping. The responsibility of the system is to ensure that the specified state mapping is applied and the change is effected at a time when the specified functions are off the stack.

Our on-line version change system is discussed in the following sections. The system has currently been implemented in C on a Sun 3/60 and has been designed for C programs. However, it can easily be extended for other procedural languages and can be ported to other systems running Unix. A description of the approach and the implementation is also given in [GJ93b].

## 4.2 System Design

The proposed approach to on-line version change is different from the traditional one of dynamic linking. The basic idea is to transfer state at an appropriate time, from a process running the old version of the software to a process running its new version.

The user is responsible for developing, compiling, linking, and debugging both the versions of the program. They must also be linked with a run-time library supplied with the system. Development of the new version is assumed to be done off-line. That is, it does not form a part of the on-line replacement system. The user is also responsible for ensuring the validity of the on-line change. Thus he must determine the set of functions which should be off the stack at the time of change using techniques discussed in Chapter 3 and specify it to the system as a parameter of the on-line change.

The major modules of the system are shown in Figure 4.1. The system presents a *modification shell* to the user. This shell accepts some commands from the user and executes them. All commands related to on-line replacement go through this shell. The commands include those for running a program, replacing it with a new version etc. The list of the commands along with a summary of their functions is given in Table 4.1.

The run module contains routines for running a user program and the replace module contains routines to replace the currently running program with a new version. The executable file generated by the linker contains symbol table information for the program. The symbol table handling module contains routines to read this information from the file and store it in the form of an internal table. It also contains routines to search the symbol table etc. The symbol table of the program is required to find the starting addresses of routines. These addresses are required to check if the stack contains any routine which has to be off the stack at the time of change. These are also required later during actual state transfer so that the program counter and the return addresses on the stack can be mapped to their new values.

The run module as well as the replace module create new processes. These processes run as child processes of the modification shell, and are managed by the

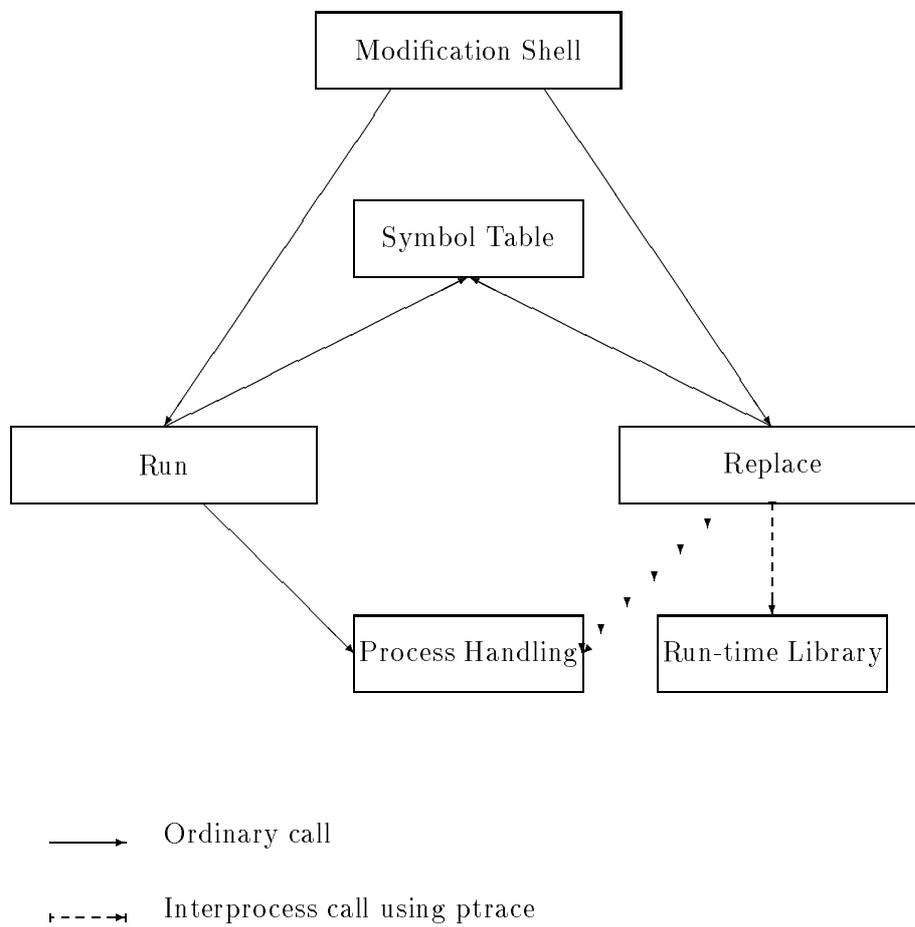


Figure 4.1: Major modules of the system

Command	Function
cd dir	Change working directory to “dir”.
help	Print this message.
kill	Kill the currently running process.
quit	Exit the modification shell.
re new-prog routines [init]	Replace the currently running program with “new-prog”, with the functions specified in the file “routines” to be off the stack at the time of change and with “init” as the initialization routine.
run prog [arg1 ...]	Run program “prog”.
tty [tty-name]	Set I/O terminal to “tty-name”.
! cmd	Shell escape.

Table 4.1: Commands of the Modification Shell

*process handling* module. The process handling module controls the two processes by means of the *ptrace* system call [Sun88]. The *ptrace* call allows a parent process to inspect or modify the address space and machine registers of its child. A running child process can be stopped by sending it a signal. The parent can resume the stopped child at any address. This mechanism is used to inspect the state of the process running the first version to determine if the stack contains any functions which are required to be off the stack at the time of change and also for actually performing the state transfer. It is also used to make a child process call some run-time library routines.

To transfer state, the process running the first version is made to call a library routine<sup>1</sup> which stores the limits of data and stack addresses in variables whose names are known to the replacement system. This is necessary because the user program may dynamically alter these limits. The system reads these values and writes them onto the corresponding variables of the process running the new version. The new process is then made to call a library routine which expands the data and stack space

---

<sup>1</sup>This routine is linked with the user program and its address is obtained from the symbol table.

to the limits specified by these values. The replace module then copies the data and stack of the first process onto the second. The machine registers are copied next. In the case of the program counter, a mapping needs to be made since the starting address of the currently executing routine might have changed. This is done by first finding which routine is currently executing and what is the offset of the currently executing instruction from the starting address of this routine. To do this, the value of the program counter is compared with the addresses of the various routines as given by the symbol table which is kept sorted on the address. The new starting address of the routine is then found out from the symbol table for the new version of the program and this is added to the offset to get the new value for the program counter. A similar procedure is carried out for all the return addresses on the stack. After the state transfer is over, the old process is killed and the new one is continued.

## 4.3 Implementation Issues

The system has been implemented on a Sun 3/60 workstation running SunOS. This section describes some of the implementation issues.

### 4.3.1 Maintaining Consistency of Pointers

When data is copied from the old to the new version, it is necessary to preserve the consistency of pointers. Our design ensures this in a very simple way. On MC68020 and other architectures using segment based addressing, the first virtual data address of a process is always on a segment boundary after the text. We assume that the text in the two versions occupies the same number of segments<sup>2</sup>. The shell checks if this condition holds and gives an error message if it does not. We also assume that no extra global or static variables have been added in the new version<sup>3</sup>. Under these two assumptions, except for addresses of local variables of changed functions, the address of any data object remains invariant across the two versions. Since the

---

<sup>2</sup>This is a reasonable assumption since the segment size is typically quite large (2MB in MC68020).

<sup>3</sup>The system does provide a way to add new global data in an indirect manner. This is discussed later.

change is installed at a time when no changed functions are on the stack (recall that with functions as the unit of change, the set of functions required to be off the stack at the time of change always includes the set of changed functions), all pointers into data point at the right places after the change. Addresses of the various routines may, however, change across versions and our system does not guarantee the consistency of pointers into text.

### 4.3.2 Deciding the Time of Change

Due to the difficulties in automatically ensuring validity as illustrated in Chapter 3, the system makes no attempt to find the set of functions which should be off the stack at the time of change to ensure validity. Thus the user must determine this set using the techniques discussed in Chapter 3 and specify it as a parameter of the on-line change. The system only ensures that the change is done when none of the specified functions are on the stack. To check if the change can be installed at some point in time, the shell unravels the run time stack of the process running the older version of the user program to find all the return addresses on the stack. These addresses and the value of the program counter are compared with addresses of various routines stored in the symbol table to determine the set of functions which are on the run-time stack. Immediately after the new process has been created and its symbol table built, the system first checks if the change can be installed. If the change can not be installed at the time of the first check, the shell finds the lowermost frame on the stack which corresponds to a function specified to be off the stack at the time of change. The return address from this frame is modified to the address of an illegal instruction which is a part of the library linked with the user program. The original return address is saved in a variable. The user process is then continued. When the last functions on the stack, which is to be off the stack at the time of change, returns, the process incurs an illegal instruction trap and the shell which has been waiting for this event to occur is notified. At this point in time, the stack is guaranteed to contain no functions which have been specified to be off the stack at the time of change. Change installation takes place at this time. Note that if a program receives an illegal instruction trap due to some other reason, the change may get initiated at

a wrong instance in time. However, for programs compiled by a correct compiler, this can never happen.

### 4.3.3 Handling Open Files

At the time of change, the user program will typically have some files open for input and output. The same files must be opened for the process running the second version at the same offsets for file I/O to work correctly. The shell process has no means of knowing which files a child process has opened. The run-time library of the system contains special versions of the `open` and `close` routines which do some book keeping also along with their normal tasks. These versions have to be linked with the user program instead of the normal C library ones. The special version of the `open` routine adds an entry consisting of file name and mode of opening (read, write, append etc.) in an array which is indexed by the file descriptors. At the time of state transfer, before copying the data of the old to the new process, the shell makes the old process execute a library routine which stores the current file offsets of all open files in their respective entries in this array. When the data space is copied, this array also gets copied to the process running the new version. The shell then makes the new process call a library routine which opens files whose names are in this array with the specified opening modes and sets the current file offsets to their respective values. The order of opening the files is such that the file descriptors are preserved. This ensures correct file I/O after the modification has been done.

### 4.3.4 Initialization Routine

The programmer can provide an initialization routine in the new version of the program which will be called by the system at the time of the change. This routine can be used for specifying any arbitrary state mapping.

The initialization routine can also be used for other purposes. For example, many programs use an array containing pointers to functions and the array is unchanged during the execution of the program. In such a situation, the initialization routine

can contain code to reinitialize these pointers since the system does not support automatic mapping of pointers to functions.

### 4.3.5 Adding and Deleting Global Data

As explained earlier, any change in global variables of the program may destroy the consistency of pointers (local variables pose no such problems and can be added and deleted freely.) But new versions may require new data structures which may also need to be initialized with state information acquired so far. The system provides the flexibility to add new global data, though in an indirect manner. To add new data, the original program must declare some extra pointer variables which are not used in the first version. The initialization routine must have code which allocates space using `malloc` or `calloc`, initializes this space and places a pointer to it in one of these pointer variables which are unused in the first version of the program. Since the initialization routine is called by the system after the data has been copied to the new process, it can use the values of other global variables to initialize the space allocated. The area can then be accessed in the new version through the pointer. The main drawback of this scheme is that it requires the programmer to declare some extra pointer variables in the first version of the program so that new data can be added in future versions. Furthermore, when variables are added in the new version, they can not be given meaningful names. Though not very elegant, this scheme will suffice for changes that require new data sparingly.

Global data deletion is not allowed in the system, as it will destroy the consistency of pointers and other addresses. So, any data that becomes redundant in the new version still must be kept in the new version. Note that this requires careful handling of string constants, as the current C compilers typically treat string constants as global data for the purposes of memory allocation (i.e. they are assigned memory in the data segment of the process, rather than on the stack). If string constants within a function are treated as local variables (as, for example, is done in Pascal), this problem will cease to exist. For the existing C compilers, utility programs can easily be written to check if deletion of functions has inadvertently deleted any string constants or not.

### 4.3.6 Another Approach to Adding Global Data

Another scheme for adding data, which is supported by the system and which addresses some of the problems of the above approach, is by declaring an array which is unused in the first version. All new variables in the second version are “assigned” addresses overlapping with the address range of this unused array, provided the total memory needed for the new variables is less than the size of the array. This scheme has the advantage that new variables can be accessed directly rather than through a pointer, as is done in the previous approach. Also, new variables can be given meaningful names. This scheme requires a translator to generate code for the new variables, to be linked with the second version, that “assigns” memory to variables from this “reserved” area. A translator for this purpose, though not yet implemented, can easily be written.

### 4.3.7 Other Restrictions

The system imposes some restrictions on user programs. Mainly the restrictions are on the use of system calls by the user program. This is because besides the data and stack, a process has some other state information also which is kept in the kernel. Since a user-level implementation such as ours can not modify this information, the user program is prohibited from making system calls which modify such information. Most of these calls can be allowed if special versions for the system calls are written which store this information in some user variables before making the system call. The value of these variables will be copied to the new process at the time of state transfer. The new process can then be made to call a library routine which makes appropriate system calls to modify the kernel information in the same way as the old process. At present, this has been done only for `read`, `write`, `open`, `close` and `chdir` system calls. This approach will not work in certain cases. For example, if the user program creates a child process and is then replaced by a new version, the new process running the second version will bear no relationship to the child process.

However, these restriction can easily be overcome in a kernel implementation of the system. Instead of copying state from one process to another, code of a process itself can be modified in such an implementation and the replace operation can be

implemented as a system call. Such an implementation will also be more efficient since the whole of the new code need not be copied in the process at the time of change but can be “faulted in” when necessary in a virtual memory system. Thus the time taken for installing the change in such a system would be independent of the size of the program.

## 4.4 An Example

Client-server type of interaction is commonly used in computer systems. In this interaction model, there is a server process which performs certain tasks on request from the clients. The server is typically a non-terminating process while the clients run for short durations. In a client-server system, installing a new version of the server in the traditional manner will imply denial of service to all the clients for the duration of the installation. This can be avoided by on-line version change.

Let us consider a simple print server that, on getting a command from the client, formats a file for printing on a particular type of printer and then prints it on the printer. This example is used to illustrate how a new version of the server can be installed on-line. For simplicity, in the example the client-server communication is done through a named pipe [Sun88] (in an actual implementation, it is often done through Unix Sockets).

In the first version of the program, the server accepts only one command from the client on the pipe which is of the form: `fp filename`. On receiving the command the server formats the file `filename`, keeping the formatted output in a temporary file, and then prints this temporary file. If the executable code of the server program is in the file `server`, then the server has to be started through the modification shell, by giving the command

```
run server [params]
```

The overall structure of parts of the server is shown in Figure 4.2. Note that variables `extra1`, `extra2`, `extra3` have been declared, though not used in the server. These are to support addition of global data, if needed.

```
char *extra1, *extra2, *extra3 ; /* Some extra pointer variables
                                   being declared for future use */

main()
{
    char infile[MAXFILENAMELEN] ;
    char outfile[MAXFILENAMELEN] ;
    FILE *fp ;

    Create command pipe fp ;
    for (;;) {
        read_command(fp,infile,outfile) ;
        format(infile, outfile) ;
        print(outfile) ;
    }
}

read_command(fp,infile,outfile)
FILE *fp ;
char *infile ;
char *outfile ;
{
    char cmd[5] ;

    fscanf(fp,"%s %s", cmd, infile) ;
    strcpy(outfile,"/tmp/server") ;
}
}
```

Figure 4.2: Print server main loop

```

read_command(fp,infile,outfile)
FILE *fp ;
char *infile ;
char *outfile ;
{
    char cmd[5] ;

    fscanf(fp,"%s %s", cmd, infile) ;
    if (!strcmp(extra1,cmd))
        fscanf(fp,extra2,outfile) ;
    else
        sprintf(outfile,"/tmp/server") ;
}

```

Figure 4.3: Version 2 of function `read_command`

In the second version, the client can give the server another command of the form: `fpo infile outfile`. On receiving this command the server formats and prints `infile` and also leaves a copy of the formatted file in `outfile`. Clearly, for this version of the server the routine `read_command` has to be modified. The formatting and printing routines will remain the same. The new version of the `read_command` function is given in figure 4.3. In this new version, if `fpo` command is given, the `read_command` function assigns to `outfile`, the name of the file specified in the command, otherwise the temporary file in the `/tmp` area is assigned, as in the first version.

The user (the person who will install the new version) has to specify the change configuration file (i.e. the file which contains the names of the functions which should be off the stack at the time of change) and the initialization routine for doing an on-line version change. It is easy to see that the new version of the function `read_command` is a functional enhancement of the old one and hence the change will be valid if the function `read_command` is off the stack at the time of change. Therefore, the change configuration file just contains:

`_read_command`

```

init_change()
{
    extra1 = malloc(4*sizeof(char)) ;
    extra2 = malloc(3*sizeof(char)) ;
    extra1[0] = 'f' ;
    extra1[1] = 'p' ;
    extra1[2] = 'o' ;
    extra1[3] = '\0' ;
    extra2[0] = '%' ;
    extra2[1] = 's' ;
    extra2[2] = '\0' ;
    cause_trap() ;
}

```

Figure 4.4: Change initialization routine for print server

The initialization routine is necessary due to the use of extra variables. The initialization routine `init_change` initializes the value of these variables, and is shown in figure 4.4. The use of this routine by the system is explained earlier. To transfer control properly, the `init_change` routine calls a system provided `cause_trap` function before exiting.

If the executable version of the new server program is kept in the file `server.new` and the, the change configuration file is named `change_config`, then the new version is installed by giving the command

```
re server.new change_config _init_change.
```

## 4.5 System Performance

Some experiments were conducted to determine the overhead incurred due to on-line replacement. The overhead can be divided into two parts — one is the overhead during normal execution when some extra activities are performed to help the replacement when required. The other is the overhead that is incurred during replacement.

In the experiments, it was found that the overhead during normal execution is minimal. A program executed through our modification shell (but with no modification done) takes approximately the same time as it does if it were executed through the Unix Shell. This is to be expected since minimal bookkeeping activities are being performed during normal execution.

The overhead during the actual replacement comes largely due to the transfer of state information, changing the addresses, and searching the stack. All of these depend on the program and the size of the stack. In our experiments we found the overhead due to these to be very small. For the example given in the previous section, where the function `read_command` was replaced with a new version, we found that on a moderately loaded Sun 3/60 system, the change takes about 200 ms (CPU time).

To test the effect of on-line version change on the performance of a running software, another experiment was conducted along the lines of the experiment described in [SF89a]. The server code was modified to send a message to a performance monitoring program after processing each request received. The performance monitoring program records the number of messages received every second. The server was then run through the modification shell and heavily loaded by running a client which continuously sends it requests. The change to the second version of the server was then made at a random time and the performance of the system during the change was observed. The following observations were made.

Rate of performing requests before the change	3.83 per second
Duration of change	1.04 seconds
Rate of performing requests during the 2 second interval containing the change	1.5 per second

The change duration of 1.04 seconds was spread over two one second intervals. During these 2 seconds, the number of requests processed was found to be 1 and 2 respectively. This clearly shows that the installation of the change did not cause a significant deterioration in the performance of the server. The rate of performing

requests after the change, may in general, increase or decrease and will depend on the difference between the old and the new versions. In this case, however, it remained the same, as there was only a minor difference between the two versions.

## 4.6 Experiments with On-line Change

Some experiment were conducted to investigate certain engineering issues involved in on-line change. The following were the major goals of the experiments.

1. To find out if the sufficient conditions given in Chapter 3 are enough in practice to ensure validity,
2. to determine how much effort is required on the part of the programmer to ensure validity and
3. to study the computational cost of installing an on-line change as a function of the size of change.

A large enough program whose source code was available and which was similar to programs for which on-line change might be required in practice was required for the purpose of the experiment. Some other restrictions were imposed on the required program because of the current limitations of the on-line change system. It was necessary the program be written in C and that it should not use any system calls which are not allowed by the change system. It was finally decided to use the game program *Nethack*.

*Nethack* is a display oriented adventure game [RT90]. The objective of the game is to descend a dungeon and find the “Amulet of Yendor”. On the way, numerous kinds of monsters are encountered which have to be killed using weapons and magic. Found also in the dungeon are various kinds of useful objects and treasures. It was decided to use this program for our experimentation since it has many features of programs for which on-line change might be desired in practice e.g. it is interactive and (potentially) long running. Its source code is also freely available. It uses some system calls which can not be handled by the system but these are not essential to

the working of the program and can be easily removed. The program is quite large (76000 lines of C code) and thus also satisfies the size requirements.

### 4.6.1 Experimental Procedure

Five different versions of the program were prepared. A bug was deliberately introduced in the base version. Versions 2 and 5 introduced new features in the game<sup>4</sup>. Version 3 removed the bug introduced in the first version and the fourth version removed another bug deliberately introduced in the new feature of version 2. Whenever a change was installed, its validity had to be ascertained. This was done by exercising the program after the change and observing its behavior. Erroneous behavior of the program after the change from an existing version to the next indicated errors in the initialization routine of the new version or in the set of functions specified to be off the stack at the time of change. After modifying these, the version before the change was reinstated and the change was attempted again. This procedure was repeated till a valid change could be observed. This exercise was undertaken for each change. Finally, when all the changes were valid, all the changes were again made many times to get the average time taken for the change. For each of the changes, the following observations were made.

1. Number of affected files ( $\Delta_{\text{files}}$ ).
2. Number of affected functions ( $\Delta_{\text{functions}}$ ).
3. Number of affected lines of code ( $\Delta_{\text{loc}}$ ).
4. Number of new functions ( $N_{\text{new fns}}$ ).
5. Number of new global variables ( $N_{\text{new vars}}$ ).
6. Space occupied by the new variables in bytes ( $S_{\text{new data}}$ ).
7. Size of initialization routine in lines of code ( $S_{\text{init}}$ ).

---

<sup>4</sup>The features were present in the original source code also but using conditional compilation, they were not included in the base version.

8. Number of functions required to be off the stack at the time of change to ensure validity ( $N_{\text{off stack}}$ ).
9. Average time taken for change in milliseconds ( $t_{\text{change}}$ ).
10. Number of attempts required for achieving a valid change ( $N_{\text{attempts}}$ ).
11. Total effort required in minutes ( $t_{\text{effort}}$ ).
12. Reasons for expecting for validity of change.
13. If the number of attempts required was more than one then why so many were required.

#### 4.6.2 Experimental Data and Observations

Observations 1 to 11 have been tabulated in Table 4.2. As can be seen, there are two small changes which affect only one file and one function each, one medium size change and one relatively large change. In an on-line change, one normally expects the changes to be small — if the change is very large, ensuring validity will be hard and hence installing the change will be difficult. Hence, very large changes are likely to be installed in a traditional manner. The difficulty of ensuring validity is also illustrated by this experiment. The “1 to 2” change, in which 42 functions were changed, required five attempts by the developer to “debug” his change and achieve validity while the other changes required only one or two attempts. The effort required for preparing for and installing the “1 to 2” change is also dramatically higher than that of the small “2 to 3” change (600 minutes v/s 5 minutes). However, as the experiment shows, the time taken by the system for installing the change is relatively independent of the size of change.

Let us now see how the validity was “ensured” by the developer. In all the cases in which functionality was added to the program, it could be easily seen that the new versions of the changed functions were functional enhancements of the old ones provided a suitable state mapping was used. This mapping gave initial values to the new variables and also mapped some variables which would have had different values

Change from version:	1 to 2	2 to 3	3 to 4	4 to 5
$\Delta_{\text{files}}$	26	1	1	6
$\Delta_{\text{functions}}$	42	1	1	3
$\Delta_{\text{loc}}$	1100	2	2	30
$N_{\text{new fns}}$	9	0	0	1
$N_{\text{new vars}}$	3	0	0	1
$S_{\text{new data}}$	3000	0	0	12
$S_{\text{init}}$	1000	225	225	225
$N_{\text{off stack}}$	42	1	1	3
$t_{\text{change}}$	704	776	716	736
$N_{\text{attempts}}$	5	2	1	1
$t_{\text{effort}}$	600	5	5	15

Table 4.2: Observations in the experiments

had the new version been run instead of the old one. This mapping was easily found out by inspection of the two versions of the program. It was verified that the control variables were not defined by any unchanged functions and the mapped variables were not used or defined by any unchanged functions. Validity then followed from Theorem 3.3. In the case of the version changes which involved removing of bugs, it was found by inspection that any state in which the changed functions were off the stack was a reachable state for the new program version even though the conditions of Theorem 3.4 were not satisfied.

Multiple attempts were required mainly because of errors or omissions in the initialization routine. A common source of error was the omission of reinitialization of arrays containing pointers to functions<sup>5</sup>. Also in the first change, the effect of the change on some variables was only noticed after a few unsuccessful attempts which increased the number of attempts at changing the initial version. This problem could probably have been alleviated if automatic tools were used to detect such dependencies.

### 4.6.3 Conclusions

The following conclusions can be drawn from the above observations.

---

<sup>5</sup>This is required in our implementation even if the change is trivial, as pointed out earlier (cf. 4.3.1.) This is also why the initialization code for even the simpler changes runs to 225 lines.

1. In most cases, it is reasonably easy for the programmer to find a state mapping which makes the change a functional enhancement. Thus, it is frequently not necessary to resort to Theorem 3.5 and it is therefore possible to avoid the associated difficulties.
2. The amount of effort required to find the state mapping, write the initialization routine etc. depends in general on the size of the change and not on the size of the program. The effort can be further reduced if some program analysis tools like tools for finding all callers of a given function, finding all uses and definitions of a given global variable etc. are available.
3. The time required by the system to install the change is independent of the size of the change and depends on the size of the program only. This was to be expected since most of the time required for change is spent in transferring state the size of which depends on the program size.
4. Especially for large changes, the change is likely to have “bugs”. It is therefore necessary, for real applications, to have a mechanism for “validating” changes. This issue is however not addressed in this thesis. Alternatively, changes should be introduced in “small units” such that each can be validated and then tried for some time.

## Chapter 5

# On-line Change for Object Oriented Programs

In this chapter, we discuss the application of our general framework to study on-line changes to object oriented programs. We first briefly review the major features which characterize object oriented programming languages.

Procedural languages support the “data-procedure” paradigm in which active procedures act on the passive data that is passed to them. In contrast, object-oriented languages employ a data or object-centered approach to programming. Instead of passing data to procedures, objects are asked to perform operations on themselves. The main program unit in an object-oriented language is the *class*. A class definition serves as a prototype for its *instances* i.e. the class provides all the information necessary to construct and use objects of a certain kind, its instances. Each instance has storage allocated for maintaining its individual state. This state is referenced by *instance variables*. There might also be a provision for maintaining some state common to all the instances of a particular class. This state is referenced by the *class variables*. The class definition also specifies the *methods* for the instances of the class. These are simply procedures which are used by objects to modify or access the state of other objects. Computation is performed by sending *messages* to objects which invoke methods of the objects. Object-oriented languages typically have four characteristics: information hiding, data abstraction, dynamic binding and

inheritance.

Information hiding is achieved by restricting the state of a module to private variables visible only from within the scope of the module and allowing only a localized set of procedures to directly manipulate this state. Information hiding is important for ensuring reliability and modifiability of the software by reducing interdependencies between software modules.

Data abstraction is a way of using information hiding. In this paradigm, abstract data types are defined which consist of an internal representation and a set of procedures used to access and manipulate the data [GH78].

Dynamic binding is used in object-oriented languages to implement polymorphism. Polymorphism allows different classes to have same method names. The actual address of the code to be executed when a message is sent is determined at run-time instead of at compile-time. This lets the programmer define generic types which can be used with arguments of many different types. Thus the same “Stack” class in an object-oriented language might be used to create objects which represent stacks of integers, strings, real numbers, etc. or even stacks which contain objects of several different types at the same time.

Inheritance allows creation of classes that are specializations of other classes. Creating a specialization of an existing class is called *subclassing*. The new class is called a *subclass* of the existing class and the existing class is a *superclass* of the new class. The subclass inherits instance variables, class variables and methods from its superclass. It may add new instance variables, class variables and methods and may redefine some of the inherited methods. Inheritance enables programmers to create new classes of objects by specifying only the differences between the new class and an existing class instead of starting from scratch each time.

Currently there is a lot of interest in the object-oriented paradigm. Several commercial and research languages exist for object-oriented programming. Some of the more popular commercial languages are C++ [Str86], Objective-C [Cox86] and Smalltalk [GR83, Xer81]. CLOS [DG87] allows object-oriented programming in Common Lisp. Some languages combine the object-oriented paradigm with concurrent computing. Examples of such languages are Actors [Agh86] and

Concurrent Smalltalk [Yok90]. Some other languages also partially support the object-oriented paradigm such as Ada [Uni80], Modula-2 [Wir85] etc. We have based our program model on Smalltalk since it seems to be the simplest language and is unencumbered with non object-oriented features.

We want to study on-line changes in the object-oriented model not just because it is a popular programming paradigm but also because it is a very good vehicle for studying changes in types. Such changes were not considered in the model for procedural languages. Changes in types of variables have a bearing both on the timing of change and on the state mapping used. In the simplest case, a type change could be a change of an integer variable to a variable of the floating point type. In the more general case, it could be changing the implementation of a queue from an array to a linked list. It is clear that in such cases, the state mapping must map the variable's value to a "corresponding" value in the new domain. Moreover, this must be done while there are no operations in progress on the variable so that it has a "consistent" value at the time of change. Since procedural languages do not support abstract data types, the code for the operations may be spread over several different portions of the program and difficult to associate with the data on which it operates. Thus it is difficult to find if any operations are in progress on a variable whose type is changed across versions except if the variable is of a simple type such as integer. In an object-oriented language, the basic units of a program are classes which normally represent abstract data types. Changes in types are therefore easier to study in this model. Further, we omitted pointers and dynamically created storage from our earlier model because data-flow analysis is very expensive and conservative if the language allows pointers. Dynamically created objects are, however, an integral part of the object-oriented paradigm but are easier to study because of the structure imposed on the dynamic structures by the class mechanism.

In this chapter, we consider a simple object-oriented program model. We describe our model of state and the set of state mappings to which we restrict ourselves. We then obtain sufficient conditions for validity of change in this model which are very similar to the ones developed for the procedural model in Chapter 3. Finally we consider some implementation issues.

## 5.1 Program Model

For simplicity, we consider a very simple program model based on Smalltalk. In our program model, a program consists of a number of class declarations. A class declaration consists of the name of a superclass, names of *instance variables* and definitions of *instance methods*. A method definition describes how an object will react to the receipt of a particular message. A method is composed of local variable declarations and statements which can be assignment statements, return statements, method invocations and the normal conditional and loop statements. A method can use the instance variables and the pseudovisible `self` which always refers to the object itself. A method is assumed to return `self` if there is no explicit return value.

The analogue of the main program in procedural languages is a `main` class here. An instance of this class is automatically created on program startup and the method `mainmethod` is invoked on it. The inheritance mechanism is same as in Smalltalk. A class inherits the variables and methods of its superclass and can optionally redefine some of the inherited methods.

There are two builtin classes, `Boolean` and `Integer` which implement boolean and integer types respectively. These are the only classes to have constant literals which can appear in the program. A predefined `nil` object can also appear as a constant literal in the program. There is also a builtin class `Object` which serves as a superclass of all other classes. The class `Object` may have various useful methods which are relevant to all objects such as methods for printing, comparing, copying etc. of objects. Each class has a builtin class method `new` which returns a new instance of the class with all instance variables initialized to `nil`.

As an example, we present a simple program to maintain a list of integers. The operations allowed on the list are to append an integer to the list, to search for an integer in the list, to delete the first occurrence of an integer in the list and to print a list. The user can give commands to invoke these operations. The program consists of two classes: `main` and `list`. The `mainmethod` of class `main` handles the user-interface and uses the `list` class to maintain the list. The classes `main` and `list` are shown in Figure 5.1 and the class `listnode` used by the class `list` is shown in Figure 5.2.

```

Class main
begin
vars l; /* the list of integers */
Method mainmethod
  vars command, param;
  begin
    l ← list.new;
    repeat
      read command and param;
      if command is "add" then
        l.add(param);
      else if command is "delete" then
        l.delete(param);
      else if command is "search" then
        if l.search(param) then
          write "found";
        else
          write "not found";
        else if command is "print" then
          l.print;
        until command is "quit";
      end
    end
  end /* of class main */

Class list
begin
vars head;
Method add (e)
  vars l,n;
  begin
    n ← listitem.new;
    n.setvalue(e);
    n.setlink(nil);
    if (head ≠ nil) then begin
      l ← head;
      while (l.getlink ≠ nil) do
        l ← l.getlink;
      l.setlink(n);
    end else
      head ← n;
    end
  end

Method delete(e)
  vars p, l, found;
  begin
    p ← nil;
    l ← head;
    found ← false;
    while l ≠ nil and not found do begin
      if l.getvalue = e then
        found ← true;
      else begin
        p ← l;
        l ← l.getlink;
      end
    end
    if found then begin
      if p ≠ nil then
        p.setlink(l.getlink);
      else
        head ← l.getlink;
      end
    end
  end

Method search(e)
  vars l, found;
  begin
    l ← head;
    found ← false;
    while l ≠ nil and not found do begin
      if l.getvalue = e then
        found ← true;
      else
        l ← l.getlink;
      end
    end
    return found;
  end

Method print
  vars l;
  begin
    l ← head;
    while l ≠ nil do begin
      write l.getvalue to terminal;
      l ← l.getlink;
    end
  end
end /* of class list */

```

Figure 5.1: The main and list classes

```

Class listnode
vars value, link;
Method setvalue(v)
  begin
    value ← v;
  end
Method getvalue
  begin
    return value;
  end
Method setlink(l)
  begin
    link ← l;
  end
Method getlink
  begin
    return link;
  end
end /* of class listnode */

```

Figure 5.2: The `listnode` class

We have permitted ourselves a degree of looseness in the syntax but the meaning of the constructs used should not require any explanation. For example, the equality test is written as an infix operator though in an object-oriented language, it would normally be written as a method call to one of the operands with the other as the parameter (test for equality with `nil` can be provided by the class `Object`.) We have also deliberately introduced a degree of vagueness in places by using the statements and expressions in *italics* so as to free ourselves from the burden of irrelevant detail.

## 5.2 Execution Model

Before formal definition of the notion of process state which is crucial to understanding on-line change, we informally describe the execution model and the state used by it. This is a simplified description of the implementation of Smalltalk described

in [Ing78].

At run time, a single process executes a program. The state of the process executing a program consists of an object space and a message stack. The object space contains the state of all the existing objects. The state of an object consists of the following.

1. Pointer to the class structure.
2. Values of the instance variables. The value of an instance variable is either an integer or boolean constant or pointer to another object. Some bits in the value may be used to distinguish between these cases.

A class structure contains the following.

1. Pointer to the superclass. The superclass is `Object` if none other has been specified.
2. A *method dictionary* which contains method name to code address mapping for the methods defined in the class.

The message stack consists of stack frames each corresponding to one unfinished message invocation. Each stack frame contains the following.

1. A return address at which execution is to resume when this method invocation is finished.
2. The *self* pointer, i.e. pointer to the object on which the method has been invoked.
3. Arguments passed to the method.
4. Values of local variables of the method. As before, the values are either integer constants or addresses of other objects.

Besides the stack and the object space, the state also has a program counter and a stack pointer. The state is symbolically represented in Figure 5.3. When a method invocation statement is executed, an appropriate stack frame is built. Then the

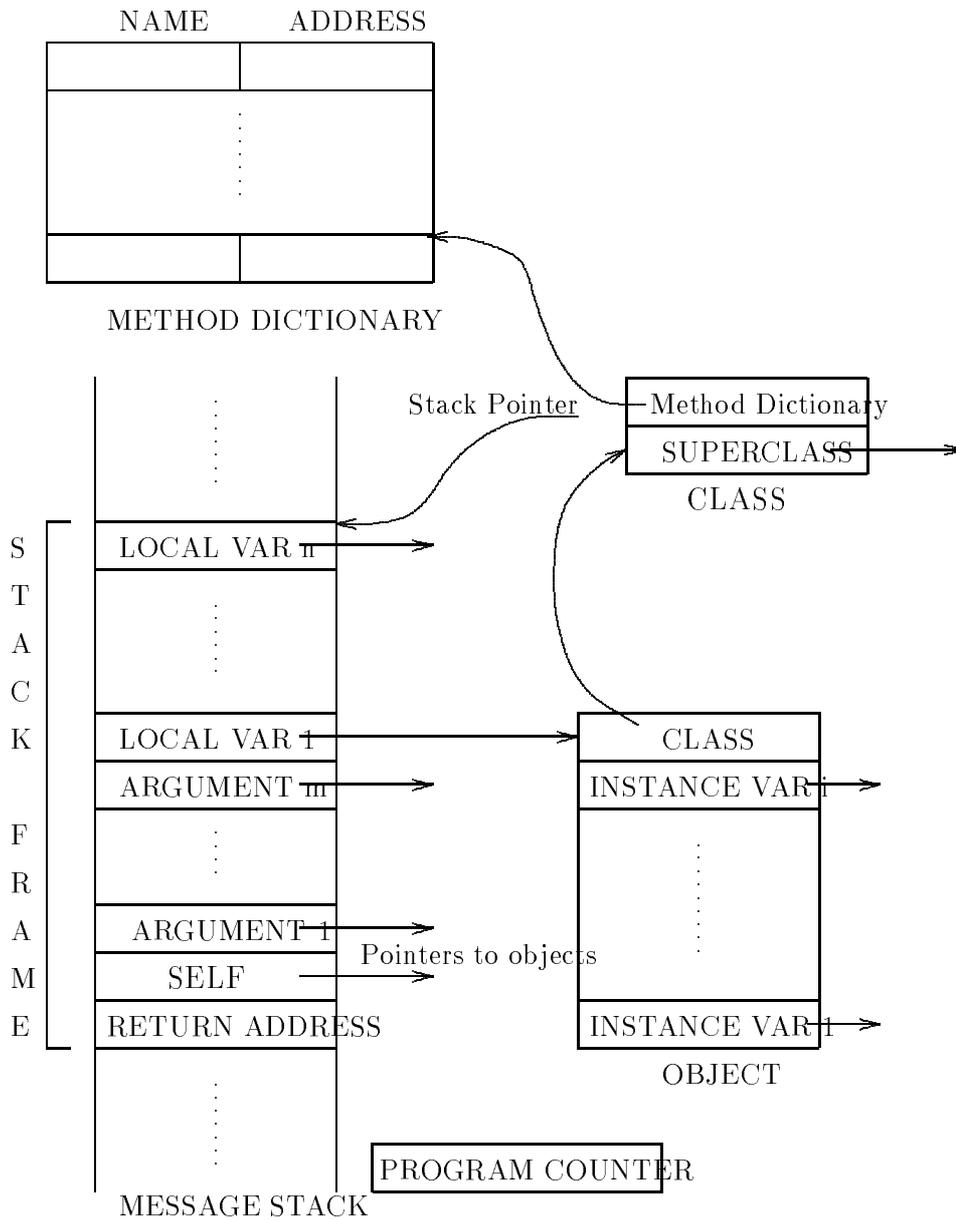


Figure 5.3: Run time state of the program

method's name is looked up in the object's class' method dictionary and if not found in the method dictionary of its superclass and so on. Then the control passes to the code thus located. On executing a return statement, the stack frame is popped, the return value pushed on the stack and control is passed to the indicated return address.

An object which is not reachable from the stack is known as a *garbage object*. Periodically, a garbage collector algorithm is run to remove such objects from the object space.

### 5.3 Model of State

For this program model, the state can no longer be modelled as a name-value mapping because of the existence of pointers. In the denotational description of programming languages (see for example [Sto77, Sch86]), this difficulty is overcome by using two mappings — a name-location mapping (usually called the *environment*) and a location-value<sup>1</sup> mapping (usually called the *store*). However, the explicit presence of locations in the state model makes it difficult to express the fact that the actual address in memory of an object is unimportant i.e. the location of an object can be changed to a new (unused) location without affecting the state, provided that all references to the object are also consistently changed.

To get around these difficulties, we model the state as a directed graph. This seems to be the most intuitively obvious model as the objects, in general, form an arbitrary graph. The stack can be modelled either as a mapping from triplets (such as the ones for the procedural model) to vertices of the graph or as special vertices of the graph itself. We prefer the latter approach because if the stack is modelled as a mapping then the graph may have unreferenced vertices (i.e. with no incoming edges) but which are not garbage. Also this approach makes the model simpler and more uniform.

In our model of the state, corresponding to each object in the state, there is a vertex in the graph which models the state. Such a vertex is called an *object vertex*

---

<sup>1</sup>The values can themselves be locations.

and is labelled by the name of the class of which the object is an instance. Base values in the state (boolean and integer values and `nil`) are modelled by *base vertices* which are labelled by their values. If the instance variable  $x$  of an object  $o_1$  points to the object  $o_2$ , then there is a directed edge labelled  $x$  from the vertex  $v_1$  representing  $o_1$  to the vertex  $v_2$  representing  $o_2$ . Similarly if  $x$  has a base value  $v$  then there is a directed edge labelled  $x$  from  $v_1$  to a base vertex labelled  $v$ . Since base values are used “by value” and not “by reference” like other objects (i.e. the statement  $x \leftarrow y$  places a *copy* of the value of  $y$  in  $x$  if  $y$  has a base value), we insist that each base vertex should have at most one incoming edge<sup>2</sup>. The stack is modelled by *stack frame vertices* which are linked together by directed edges labelled “last frame”. Thus, there is a directed edge labelled “last frame” from vertex  $v_1$  to  $v_2$  if the stack frame corresponding to the vertex  $v_1$  is just above the one corresponding to  $v_2$  in the stack. There is also a directed edge labelled “self” from a stack frame vertex to the object vertex which corresponds to the object on which the method corresponding to the stack frame has been invoked. For each local variable or parameter  $x$  of this method, there is a directed edge labelled  $x$  from the stack frame vertex to the object or base vertex corresponding to the value of the variable. The program counter and the return addresses are modelled as labels of the stack frame vertices. The label of each such vertex is of the form  $C:m:x$  where  $x$  is some offset within the code of the method  $m$  of the class  $C$  corresponding to the particular stack frame. Thus, the label of the stack frame vertex with no incoming edge (corresponding to the topmost stack frame) represents the current program counter while the labels of the other stack frame vertices represent the various return addresses.

As an example of state, Figure 5.4 shows a part of the state of the example program given earlier. In the state shown, the list contains the elements 5, 2 and 1 and the process is currently inside the method `print` of class `list`. In the state shown, the value 5 has been printed and the value 2 is about to be printed. Each node and edge carries its label with it.  $x_1$  and  $x_2$  are assumed to be suitable offsets within the code of the methods `mainmethod` and `print` respectively.

It is clear that isomorphic graphs must be considered to represent the same

---

<sup>2</sup>This need not be done for `nil` values since “sharing” of `nil` can not make any difference to the program behavior as no operations are possible on it.

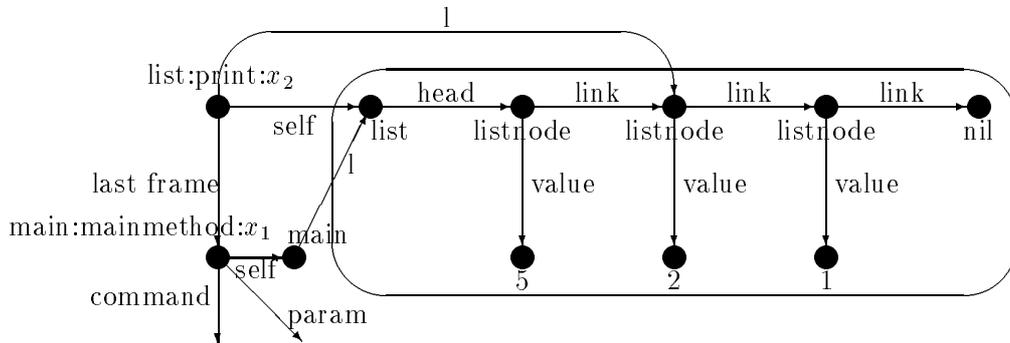


Figure 5.4: A state of the example program

state. The standard definition of graph isomorphism [Deo90] can be easily extended to handle labelled graphs as follows.

**Definition 5.1** Two labelled directed graphs  $G$  and  $G'$  are said to be *isomorphic* if there exist one to one and onto mappings  $\mathcal{V}$  and  $\mathcal{E}$  from vertices of  $G$  to vertices of  $G'$  and from edges of  $G$  to edges of  $G'$  respectively such that for all vertices  $v$  and edges  $e$  of  $G$ , the following properties hold.

1.  $v$  and  $\mathcal{V}(v)$  have the same label.
2.  $e$  and  $\mathcal{E}(e)$  have the same label.
3. if  $e$  is from vertex  $v_1$  to  $v_2$  in  $G$  then  $\mathcal{E}(e)$  is from vertex  $\mathcal{V}(v_1)$  to  $\mathcal{V}(v_2)$  in  $G'$ .

## 5.4 State Mappings

In this section, we describe the class of state mappings to which we restrict ourselves. The major difference between on-line change for a procedural program and for an object oriented program is that the identity mapping does not usually work in the object oriented case since in object oriented languages, data and the code which operates on it are closely tied together and a change in one usually entails a change in the other. So at the very least, one must do some restructuring of the objects of

the changed class. This restructuring mapping should map the state of an object of a changed class to a new state relevant to the new version of the class. The state of an object can be characterized by its *object space* defined below.

**Definition 5.2** The *object space*  $OS(s, o)$  of an object vertex  $o$  in the state  $s$  is the subgraph of  $s$  formed by  $o$  and all the vertices of  $s$  reachable from it.

Intuitively, the object space of an object  $o$  is the set of all objects which can be reached from  $o$  by following pointers. In Figure 5.4, the subgraph enclosed by the oval is the object space of the object labelled “list”.

Thus, the process of restructuring involves replacing the object space of the object being restructured by another graph. In general, the object space of an object  $o$  may have vertices other than  $o$  with incoming edges from outside the object space. Therefore, to “replace” the old graph with the new one, it is necessary to map these vertices to vertices in the new graph so that these edges can be properly mapped. The set of such objects is called the *accessible set* of the object space of  $o$  and is defined below.

**Definition 5.3** The *accessible set*  $AS(s, g)$  of a subgraph  $g$  of a state  $s$  is the set of vertices of  $g$  which have an incoming edge in  $s$  from a vertex not in  $g$ .

In Figure 5.4, the accessible set of the object space of the vertex labelled “list” is the set containing this vertex and the the second vertex labelled “listnode” (with value 2). These are the only vertices which have incoming edges which cross the boundary of the oval.

Edges from outside  $OS(s, o)$  to objects (other than  $o$ ) in  $AS(s, OS(s, o))$  signify references to objects in the object space of  $o$  from “outside”  $o$ . These thus form potential ways to modify the “value” of  $o$  without invoking a method of  $o$  itself.

Now that we have captured the state of an object and the references to it from the rest of the state, restructuring can be done by replacing the old object state by a new value and mapping the external references to the object to references within the new object. For describing this process formally, we now define a restructuring mapping.

**Definition 5.4** A *restructuring mapping*  $\mathcal{R}$  for a class  $C$  of  $\Pi$  is a pair of functions  $(\mathcal{F}_1, \mathcal{F}_2)$  where  $\mathcal{F}_1$  is a function which maps the object space  $\text{OS}(s, o)$  of an object  $o$  of class  $C$  to another graph and  $\mathcal{F}_2$  maps each vertex in  $\text{AS}(s, \text{OS}(s, o))$  to some vertex of  $\mathcal{F}_1(\text{OS}(s, o))$ .

A state mapping can be derived from the restructuring mappings for each of the changed classes as follows.

**Definition 5.5** The *state mapping*  $\mathcal{S}$  derived from the restructuring mappings  $\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_n$  for classes  $C_1, C_2, \dots, C_n$  respectively is defined as follows. For a state  $s$ ,  $\mathcal{S}(s)$  is obtained by performing the following transformation for each vertex  $o$  labelled “ $C_i$ ” ( $1 \leq i \leq n$ ). If  $\mathcal{R}_i$  is  $(\mathcal{F}_{i_1}, \mathcal{F}_{i_2})$  then  $g = \text{OS}(s, o)$  is replaced by  $\mathcal{F}_{i_1}(g)$  and each edge labelled  $l$  in  $s$  from a vertex  $n_1 \notin g$  to a vertex  $n_2 \in g$  is replaced by an edge labelled  $l$  from  $n_1$  to  $\mathcal{F}_{i_2}(n_2)$ .  $\mathcal{S}$  is defined only for states  $s$  such that there is no incoming edge labelled “self” in  $s$  to a vertex in  $\text{AS}(s, \text{OS}(s, o))$  where  $o$  is a vertex labelled “ $C_i$ ” ( $1 \leq i \leq n$ ) in  $s$ .

The restriction on the states for which the mapping is defined informally means that no methods of objects of any changed classes are in execution. Further, no methods of any objects in the object space of an object of a changed class should be in execution. This is because if such a method is currently executing then the “value” of the object may not be “consistent”. For example, during addition or deletion of an element from a doubly linked list, the pointers within the list may be temporarily inconsistent and in such a state, the object would not represent any valid list.

Note that it is possible for object spaces of two or more objects of changed classes to be overlapping. In such a situation, we require that the order in which the restructuring is performed should not affect the result.

It can be seen that such state mappings are sufficient to handle most practical situations. We therefore restrict ourselves to such state mappings only.

## 5.5 Ensuring Validity

We now give conditions to ensure validity of an on-line change in the object-oriented program model. We treat classes as the units of change. For object-oriented

programs, a class is the most natural unit of change since it typically represents a single unit of the program. This means that we consider the whole class to be changed even if only a small part of it is actually changed and that we do not “look inside” a class. By an argument similar to the one in Section 3.7 where functions were treated as the units of change, it is easy to see that the methods of a changed class must not be executing at the time of change if the change is to be installed. This restriction on the timing of change is also implied by the restrictions on the states for which the state mapping is defined and holds for all the treatment below.

We assume that classes are only changed and not added or deleted. The addition and deletion of classes can be easily handled by considering the added (deleted) class to be in the old (new) version of the program also but without being used.

If a class is changed then we consider all its subclasses to be also correspondingly changed. Thus for the purpose of determining validity, we consider the changes to the full class definition of each class (which includes the features inherited from the superclasses.) This allows us to treat changes in the class hierarchy as just changes in the definition of some classes.

As in the case of procedural languages, one can think of two possible ways of ensuring validity; one is to ensure that the state at the time of change (after a suitable mapping) is a reachable state of the new program and the second is to allow change in an unreachable state for the new program but which always leads to a reachable state. We consider the former possibility first.

### 5.5.1 Change in a Reachable State of $\Pi'$

Clearly, validity will be ensured if the change is done in a state  $s$  such that  $\mathcal{S}(s)$  is a reachable state of  $\Pi'$ . Thus, we wish to place restrictions on the type of change, the state mapping and the time of change so that this condition can be satisfied. Again, this will mostly be true for cases in which the behavior of the two versions of the changed class is closely related to each other. Analogous to the notion of functional enhancement in procedural languages, we can define the notion of enhancement of a class as follows.

**Definition 5.6** A class  $C'$  is said to be an *enhancement* of  $C$  with respect to the state mapping  $\mathcal{S}$  if for any reachable state  $s_1$  of  $\Pi$  for which  $\mathcal{S}(s_1)$  is defined and in which the statement to be executed next is of the form  $x := a.m(x_1, x_2, \dots, x_n)$  where  $a$  refers to an instance of class  $C$  or an object in  $\text{AS}(s_1, \text{OS}(s_1, o))$  where  $o$  is an instance of  $C$ , the following holds. Let  $s_2$  be the state obtained after this statement has been executed (i.e. the method  $m$  has been completely executed.) Clearly  $\mathcal{S}(s_2)$  is also defined. Then the state  $s'_2$  obtained by running  $\Pi'$  from  $s'_1 = \mathcal{S}(s_1)$  till the method invocation is complete can be the same as  $\mathcal{S}(s_2)$  for some inputs. The same holds if the statement to be executed next is  $x := C.\text{new}$ .

The definition is very similar to the definition of functional enhancement for functions (cf. Definition 3.7). Note that whether a new version of a class is an enhancement of its old version or not depends also on the state mapping used. Thus even if the two classes are apparently very similar, an appropriate state mapping must be used in order that the new version is an enhancement of the old one.

Analogous to the procedural case, the on-line change is valid if changes to all the classes are enhancements and the changed classes are not executing at the time of change. The following theorem states this formally.

**Theorem 5.1** *If  $s$  is a reachable state of  $\Pi$ ,  $\mathcal{S}(s)$  is defined and for each changed class, the new version is an enhancement of the old one with respect to  $\mathcal{S}$  then  $\mathcal{S}(s)$  is a reachable state of  $\Pi'$ .*

**Proof:** We prove the result by induction on the sequence of states (starting with  $s_{\Pi_0}$ ) through which  $s$  has been reached. In the base case,  $s$  is  $s_{\Pi_0}$  itself. Since the **main** class is assumed to be unchanged,  $\mathcal{S}(s_{\Pi_0})$  is defined and can be easily seen to be the same as  $s_{\Pi'_0}$ , the initial state of  $\Pi'$ .

We assume, as the induction hypothesis, the result to be true of a state  $s_1$  and in the induction step, show it to be true for the first state  $s_2$  reached from  $s_1$  such that  $\mathcal{S}(s_2)$  is defined. There are two cases to consider. In the first case, the statement to be executed next in  $s_1$  is of the type described in Definition 5.6. Let  $s_2$ ,  $s'_1$  and  $s'_2$  be as in Definition 5.6. Clearly,  $s_2$  is the first state reached from  $s_1$  for which  $\mathcal{S}(s_2)$  is defined. Since by assumption,  $s'_1$  is reachable for  $\Pi'$ , the reachability for  $\Pi'$

of  $s'_2$  ( $= \mathcal{S}(s_2)$ ) follows from the definition of enhancement. In the second case, the statement to be executed next in  $s_1$  is not of the above type. In this case, it is easy to see that the next state  $s'_2$  reached from  $s'_1 = \mathcal{S}(s_1)$  is the same as  $\mathcal{S}(s_2)$  and the result follows since  $s'_1$  is assumed to be reachable for  $\Pi'$ . ■

The above theorem states that in case of enhancement of classes, the change will be valid if effected in a state for which the state mapping is defined. In other words, if the change is effected when no method of any object in the accessible set of the object space of an object of a changed class is executing then the change is valid.

Determining that the change to a class is an enhancement is not easy but there are many common situations in which this property holds. One case in which the new version of a changed class is an enhancement of its old one is when the two versions of a changed class implement the same abstract data type and the restructuring mapping used for this class maps the object to the new concrete representation of the same abstract value [Hoa72]. For example, consider a new version of the `list` class of our example program. In this version, a new instance variable `tail` is used to keep track of the end of the list so that addition of elements to the list can be done in constant time. The methods `add` and `delete` are accordingly modified. The changed portions of the class are shown in Figure 5.5. Clearly, the two versions of the `list` class implement the same abstract type. The restructuring mapping that we use to make this change on-line should be such that the object space of an object of this class, before and after the mapping, should represent the same abstract value. Thus, if in the state just before the change, the object space of a `list` object represents the list “5,2,1” then the graph obtained from mapping this object space should represent the same list (according to the new representation).

Another case in which the change to a class is an enhancement is when the new version of a changed class does not implement the same abstract type as the old one but its semantics are an “enhanced version” of those of the old one. This notion can be formalized using the notion of subtypes and supertypes introduced by Leavens [Lea91]. A type  $T$  is said to be a supertype of  $T'$  if every abstract value of  $T'$  “simulates” an abstract value of  $T$ . For example, the abstract value  $[1, 3]$  of an integer interval type simulates the abstract value  $\{1, 2, 3\}$  of an integer set type.

```

vars head, tail;
Method add (e)
  vars l;
  begin
    l ← listitem.new;
    l.setvalue(e);
    l.setlink(nil);
    if tail ≠ nil then
      tail.setlink(l);
    else
      head ← l;
      tail ← l;
    end
  end

Method delete(e)
  vars p, l, found;
  begin
    p ← nil;
    l ← head;
    found ← false;
    while l ≠ nil and not found do begin
      if l.getvalue = e then
        found ← true;
      else begin
        p ← l;
        l ← l.getlink;
      end
    end
    if found then begin
      if p ≠ nil then
        p.setlink(l.getlink);
      else
        head ← l.getlink;
      if l.getlink = nil then
        tail ← p;
      end
    end
    return self;
  end
end

```

Figure 5.5: Changes to the `list` class

The formal definition of the simulation relation can be found in [Lea91].

Now suppose, we replace a class  $C$  representing type  $T$  by a class  $C'$  representing type  $T'$  which is a supertype of  $T$ . If we use a restructuring mapping which maps the object space of an object to the new concrete representation of the abstract value (of  $T'$ ) which is simulated by the abstract value (of  $T$ ) represented by the old object space, then the new version of the class can be shown to be an enhancement of the old one.

A supertype can also be replaced by a subtype if each element of the supertype is simulated by an element of the subtype. In this case, the restructuring mapping maps an object of the class representing the supertype to an object of the class representing the subtype which simulates it. This type of change can also lead to program enhancement, for example, replacing windows by bordered windows. It can be easily seen that if the restructuring mapping maps an instance of the supertype to an instance of the subtype that simulates it, the new version of the class is an enhancement of the old one.

As in the case of the function based model, Theorem 5.1 can be generalized to state that if there is a set of classes (which may include some unchanged classes as well) such that the new version of each class in this set is an enhancement of its older version, objects in the accessible set of objects of these classes are not on the stack at the time of change and all other changed classes can only be accessed (directly or indirectly) through these classes then the on-line change is guaranteed to be valid. As an example of the use of this generalization, consider a new version of the `list` class implemented as a doubly linked list of `listnode` objects. Both the classes `list` and `listnode` have to be modified to make this change. In this case, the change to the `list` class is an enhancement while the `listnode` class is only used by the class `list`. In this case, it is easy to see that having methods of these classes off the stack suffices to ensure validity provided an appropriate state mapping is used. In the object-oriented model it can not be automatically determined whether a class uses another class because of dynamic binding. This information must be determined by the programmer using knowledge about the program.

### 5.5.2 Change in an Unreachable State of $\Pi'$

The above approach clearly restricts the types of changes that can be allowed. In the general case, the state in which the new version of the program starts executing is not a reachable state for it but is guaranteed to lead to a reachable state. In the corresponding case in the procedural case, we had the condition that all variables affected by the change get “redefined eventually”. In the object-oriented case, the analogous condition is that all objects affected by the change either eventually get redefined or become garbage. To determine which objects are affected by the change, we divide the set of methods of a class into *constructor* and *value* methods [GH78].<sup>3</sup> The constructor methods only modify an object and only return `self`. The value methods return some value other than `self` besides possibly modifying the object. We assume that no method of any class modifies any of its argument objects (by sending it a message). This is a reasonable assumption as violating it would be against the philosophy of object-oriented programming.

We now try to find the classes whose objects might have been affected by the change from  $\Pi$  to  $\Pi'$ . We call the set of all such classes the *affected set*. Clearly the classes which have changed across versions but whose new versions are not enhancements of the old ones are in this set. Now consider an object  $o$  that invokes a value method of an object  $o_1$  or of an object  $o_2$  in the accessible set of  $o_1$  where  $o_1$  is an object affected by the change. Clearly, the result returned to  $o$  may also be affected by the change and because  $o$  uses this result, it itself may get affected by the change. Note that this would not be the case if  $o$  only invoked a constructor method of  $o_1$  or  $o_2$ . Thus, we include in the affected set, all classes whose methods can invoke a value method of some class already in the affected set or of an object in the accessible set of an instance of a class already in the affected set. Note that if an object is affected by the change, we implicitly treat all objects in its object space to be also affected although we do not include the classes of such objects in the affected set.

---

<sup>3</sup>The constructor are used here in a slightly different sense than in [GH78] where constructor methods are those using only which all instances of the data type can be generated.

We now prove that the affected set that we have defined is indeed a safe approximation of the actual set of objects affected by the change. Finding the exact set is of course undecidable.

**Notation** We use  $s_1 \approx s_2$  to mean that the subgraphs of  $s_1$  and  $s_2$  consisting of all vertices not in the object space of a vertex labelled “ $C$ ” where  $C$  is a class in the affected set as defined above, are isomorphic

**Lemma 5.1** *For any state  $s$  of  $\Pi$  for which  $\mathcal{S}$  is defined and no method of any class in the affected set of classes is on the stack, there exists a reachable state  $s'$  of  $\Pi'$  such that  $\mathcal{S}(s) \approx s'$ .*

**Proof:** The result is again proved by induction on the sequence of states (starting with  $s_{\Pi_0}$ ) through which  $s$  is reached. In the base case,  $s$  is  $s_{\Pi_0}$ , the initial state of  $\Pi$ . If the `main` class is in the affected set, the result is trivially true since in no reachable state of  $\Pi$  the required condition on state is satisfied. We therefore assume that `main` is not in the affected set. Also since the `main` class is assumed to be unchanged,  $\mathcal{S}(s_{\Pi_0})$  is defined and is same as  $s_{\Pi'_0}$ , the initial state of  $\Pi'$ .

We assume, as the induction hypothesis, the result to be true of a state  $s_1$  and in the induction step, show it to be true for the first state  $s_2$  reached from  $s_1$  such that  $\mathcal{S}(s_2)$  is defined and no method of a class in the affected set is on the stack in  $s_2$ . There are three cases to consider.

In the first case, the statement to be executed next in  $s_1$  is an invocation of method  $m$  on an object of a changed class  $C$  which is not in the affected set and  $s_2$  is the state just after the completion of  $m$ . In this case, the new version of  $C$  is an enhancement of the old one. This means that  $\mathcal{S}(s_2)$  is obtained after the completion of the method if  $\Pi'$  executes from  $\mathcal{S}(s_1)$ . By the induction hypothesis, there exists a reachable state  $s'_1$  of  $\Pi'$  such that  $\mathcal{S}(s_1) \approx s'_1$ . Let  $s'_2$  be the state obtained after the completion of the  $m$  if  $\Pi'$  executes from  $s'_1$ . It can be easily seen that in going from  $s'_1$  to  $s'_2$  and from  $\mathcal{S}(s_1)$  to  $\mathcal{S}(s_2)$ , exactly the same transformation is effected on the subgraph of the state consisting of vertices not in the object space of an object of a class in the affected set. It follows that  $s'_2 \approx \mathcal{S}(s_2)$ .

In the second case, the statement to be executed next in  $s_1$  is the invocation of the method  $m$  on an object of a class in the affected set. Let  $s_2$ ,  $s'_1$  and  $s'_2$  be

defined as before. It is clear that in going from  $s_1$  to  $s_2$  and from  $s'_1$  to  $s'_2$ , only the object spaces of objects of classes in the affected set are modified. It follows that  $s_1 \approx s_2$  and  $s'_1 \approx s'_2$ . Also, by assumption,  $\mathcal{S}(s_1) \approx s'_1$  and  $s_1 \approx s_2$  implies that  $\mathcal{S}(s_1) \approx \mathcal{S}(s_2)$ . It follows that  $\mathcal{S}(s_2) \approx s'_2$ .

In the third case, the statement to be executed next is of none of the above two types. In this case,  $s_2$  is the state obtained immediately after  $s_1$  and if the statement is executed in state  $\mathcal{S}(s_1)$ , the state obtained is  $\mathcal{S}(s_2)$ . Again, as in the first case, in going from  $s'_1$  to  $s'_2$  and from  $\mathcal{S}(s_1)$  to  $\mathcal{S}(s_2)$ , exactly the same transformation is effected on the subgraph of the state consisting of vertices not in the object space of an object of a class in the affected set. The result follows. ■

Informally, the above lemma states that for a reachable state of  $\Pi$  in which no instance of a class in the affected set is on the stack, the mapped state is same as a reachable state of  $\Pi'$  except for objects of classes in the affected set. Clearly, if the change is installed at such a time and all instances of classes in the affected set are guaranteed to be redefined or become garbage, the change will be valid. But this introduces additional constraints on the time of change which is undesirable. We wish to be able to install the change at any time when the state mapping is defined. The following theorem proves that this can be done.

**Theorem 5.2** *For each reachable state  $s$  of  $\Pi$  for which  $\mathcal{S}(s)$  is defined,  $\Pi'$  executing from state  $\mathcal{S}(s)$  will reach a state  $s'$  such that there exists a reachable state  $s''$  of  $\Pi'$  such that  $s' \approx s''$ .*

**Proof:** Executing  $\Pi$  from  $s$ , let  $s_1$  be the first state obtained which satisfies conditions of Lemma 5.1. This will be the state obtained after all the methods of affected classes which were on the stack in  $s$  have finished executing. Correspondingly, let  $s'$  be the state obtained by executing  $\Pi'$  from  $\mathcal{S}(s)$  till all the methods of affected classes on the stack in  $\mathcal{S}(s)$  have finished executing. Let  $s'_1 = \mathcal{S}(s_1)$ . Clearly both the computations described above affect only the object spaces of objects of classes in the affected set and therefore  $s'_1 \approx s'$ . Also, by Lemma 5.1, there exists a reachable state  $s''$  of  $\Pi'$  such that  $s'_1 \approx s''$ . Therefore,  $s' \approx s''$ . ■

Thus if the change is installed in any state for which the state mapping is defined, after the change the program is guaranteed to reach a state which differs from a

reachable state only in the object spaces of instances of the classes in the affected set. It can be easily shown, that this condition will continue to hold thereafter. If we further assure, that instances of all affected classes ultimately become garbage or suitably redefined, validity will clearly be implied. We now define what we mean for an object affected by the change to eventually get redefined. We first define the set of instance variables of an object affected by the change. For a class which is changed but whose new version is not an enhancement of the old one, this is the set of all its variables. For other classes in the affected set, this set includes the variables containing the result of an invocation of a value method on a class in the affected set and variables whose values depend on the values of some variables in the set in a computation in one of the methods. For each method, similarly a set of local variables affected by the change can be defined. An object eventually gets “redefined” if for each of its variables  $x$  affected by the change,  $x$  eventually gets redefined to a value independent of the values of variables in the set of the object’s affected variables.

As in the procedural case, from the time of change to the time when the state becomes reachable for the new version of the program, the code of the program is working on data for which it has not been designed and may be prone to such fatal errors as zero divide etc. or may get stuck in an infinite loop. That this does not happen was an implicit assumption in the proof of Lemma 5.1 and Theorem 5.2. Such crashes must, therefore be avoided if validity is to be ensured. The strategies for avoiding them are much the same as in the procedural case. There is however an additional factor of concern here. For a changed class whose new version is not an enhancement of the old one, the restructuring function should be defined so that the mapped objects “correspond approximately” to the old objects.

## 5.6 Implementation Issues

In this section, we discuss some issues in the implementation of an on-line software version change system for object-oriented programs. The main difference from an implementation for a procedural language is that of the complexity involved in

restructuring the objects.

In the object-oriented case, the state mapping is likely to be much more complex than for the procedural case. In the procedural case, one just had to map global variable values and possibly initialize some new variables. A user-supplied function was therefore sufficient for specifying this mapping. In the object-oriented case, however, all instances of the changed classes have to be restructured and it is not possible to describe the mapping by means of a single user-supplied code fragment. Therefore, instead of describing the state mapping itself, the user can be allowed to supply a restructuring method for each changed class. Thus the restructuring mapping is described for each changed class and the state mapping used is the one derived from these restructuring mappings. The specification of the restructuring method for a single class is also, however, not without difficulties. This is because the restructuring code depends on both the old and the new versions of the class. There are thus the following three possibilities for specifying the restructuring method.

1. The restructuring code uses the binary image of the old object. This is clearly not a very good idea since the restructuring code would then depend on the compiler and implementation.
2. The code uses the old instance variables to construct the corresponding new one. Some syntactical construct needs to be designed to differentiate between the old variables and the new variables of the same names. In general, this is also not a very attractive alternative as the restructuring code will depend on the actual implementation in the first version of the abstract data type which the class is supposed to implement. This is also opposed to the object-oriented philosophy since the code of a method of one class (the new version of the changed class) is directly accessing the instance variables of another class (the old version).
3. The code uses the old methods to extract information about the old instance and uses it to build the new one. This seems to be the best alternative in light of the drawbacks of the other two. Again, some syntactic constructs need to be designed to differentiate between the old and the new methods of the same

names. This is also the scheme chosen by Goullon et. al. in the DAS operating system [GIL78].

Note that the chosen scheme (3) requires that the old code of the changed classes be available during state mapping. This clearly rules out the implementation approach described in Chapter 4. One possible approach is that instead of replacing the whole code, just the method dictionary of the changed classes can be changed. The new code of a changed class can reside in a new dynamically created area and the new method dictionary can point to this area. Since we require both the old and the new code for the class, the old dictionary and code must be retained till the restructuring is over. The class structure is therefore, required to hold pointers to two method dictionaries, the current and the old. If the old dictionary is `nil`, it signifies that the class has not changed across versions. Thus at the time of change, the system moves the current dictionary pointer to the old dictionary pointer, creates a new dictionary for the new code and places a pointer to it in the slot for the current dictionary. When the restructuring is over, the space occupied by the old dictionary and code can be freed.

The restructuring of an object creates a new object which corresponds to the old one. The problem now is that all other objects referring to old objects must now refer to this new one. Thus one must find all references to the old object and replace them by references to the new object. Again there are three possibilities for achieving this. The first is to search the object space for all references to the object, which is clearly prohibitive in terms of cost. The second is to maintain for each object, the list of references to it. This means an additional cost in terms of both time and space even during normal (i.e. without change) execution of the program. The third alternative is to place a pointer to the new object in the old object itself. The entry code for each method would then have to follow this pointer if not `nil` and invoke the method on the new object thus found. Although this scheme also introduces some additional time and space cost during normal execution, the overhead is considerably lesser than for the previous scheme.

The same problem is faced in actual implementation of restructuring as well. The restructuring mapping requires to map objects in the accessible set of the object space

of the old object to objects in the object space of the new object. To implement the corresponding state mapping, all incoming edges to such objects must be mapped. A similar scheme as above can be used to handle this situation.

Another similar problem is to find at the time of change, all instances of a changed class so that they can be restructured. Again, maintaining a list of all instances of a class or searching the object space at the time of change seems too expensive. One solution to this problem is to not restructure at all at the time of change but to restructure an object whenever it is first used after the change. This approach also serves to amortize the cost of restructuring, which may be quite high in practice, and thus allows the change to be installed faster. To implement such a scheme, the entry code for each object should first find out if restructuring needs to be done and if yes, invoke the method on the restructured object. A drawback of this scheme is that the first version code of the changed classes can not be discarded after the change. Another problem is that if a second change is installed after installing the first change, there may be some objects whose state still corresponds to the first version. If such an object is used after the second change, it needs restructuring twice, once from the first to the second version and once from the second to the third version. If the class of this object has not changed from the second to the third version, this is not a problem. Also if the class was enhanced both times, the restructuring code for the third version can work with the first version representation of the object and the corresponding code since the semantics of the class do not change. But if this is not the case, the object must be restructured at the time of the second change. This will again involve a search of the run-time state for all such objects. The only reasonable solution to this problem seems to be that the code of all versions of a class should be kept and if an object is several versions out of date, then the latest method for restructuring should call the previous one and so on. This is the scheme that has been used by Fabry [Fab76].

## 5.7 Conclusions

In this chapter we have studied on-line changes to object-oriented programs. A simple object-oriented program model has been considered and a model of state defined for it. Treating classes as the units of change, the development of sufficient conditions for validity has been done on the same lines as in the sequential model where functions were treated as the units of change. Thus analogous to functional enhancement, the enhancement of a class has been defined and the result about validity that follows is also analogous to that in the procedural model. Similarly analogous to the condition that “all affected variables should ultimately be redefined”, we here obtain the condition that “all affected objects should ultimately be redefined or become garbage”.

The main difference between the object-oriented and procedural models is that because in the object-oriented case the code is closely associated with the data on which it operates, it is possible to specify more precisely the nature of the state mappings that are likely to be used in practice. Similarly it is possible to determine the data that may be affected due to changes to specific portions of the code without performing data-flow analysis. However, because of dynamic creation of objects, the state mapping is more difficult to specify and implement. Thus the simple implementation approach proposed for procedural languages does not scale up to the object-oriented case.

## Chapter 6

# On-line Changes to Distributed Programs

In this chapter, we show how our framework for modelling on-line version changes can be extended to handle distributed programs. A distributed system consists of a set of processes which communicate with each other using some method of inter-process communication. There are several types of changes possible for such a system. In [PH91], Purtillo et. al. have identified the following three general types of changes.

1. **Structure** In this type of change, the system's logical structure is changed. Processes may be added or removed and the interconnections between processes may be changed. However, the code of all processes which are in both the versions of the system remains the same.
2. **Code of processes** The system's logical structure remains the same but an alteration is made to the code of one or more processes.
3. **Geometry** In this type of change, both the logical structure of the system and the code of individual processes stay the same but it is desired to map the system on to a real distributed architecture in a different way. Geometric re-configuration is useful for load balancing, software fault tolerance and adaption to changes in available communication resources etc.

It can be easily seen that changes in geometry involve only implementation issues. Geometric reconfiguration has been considered in the form of process migration in [AF89, Che88]. These implementations support the migration of a process to another processor of the same architecture and running the operating system. The Polyolith software bus system [PH91] supports geometric reconfiguration across processors of different architectures. Architecture independence is achieved in this system by modelling a process as an implementation of an abstract data type and capturing its state in the form of an abstract state which is common for all architectures.

Dynamic structural changes to a distributed system are supported by several systems e.g. Conic [KM85], Durra [BDW90], Polyolith [PH91] etc. A formal framework for modelling such changes was given by Kramer et. al. in [KM90]. They gave a definition for *node quiescence* and claimed that if a change is made at a time when all the involved processes are quiescent, the system is in a “consistent state” after the change. Consistency of state was however not formally defined and no proof was offered for the claim.

On-line changes in code of individual processes are supported by the Polyolith [PH91] and the Argus [Blo83] systems. However these systems only allow modules to be replaced by new versions which are very similar in behavior to the old ones (see Chapter 1 for details).

In this chapter, we present a framework for supporting changes to codes of individual processes of a distributed program. The framework uses the most general model of inter-process communication, namely unrestricted message passing. We develop sufficient conditions for validity of an on-line change in this model. The conditions are general and apply for both the synchronous and asynchronous models of message passing.

## 6.1 Execution Model and Framework

Following [CL85], we consider a distributed system  $S$  to consist of a set of *processes* and a finite set of *channels*. It is described by a labeled, directed graph in which

the vertices represent processes and the edges represent channels. Processes communicate through asynchronous or synchronous message passing. Each process has a unique *name* associated with it. In the case of asynchronous communications, we assume that the messages sent from one process to another arrive in the order in which they were sent, but messages sent by different processes to a process may arrive in any order.

In the case of asynchronous message passing, the channels are assumed to have an infinite buffer capacity and zero capacity in the case of synchronous message passing. In the asynchronous case, the state of the channel is the sequence of messages sent along the channel, excluding the messages received along the channel. Each process is defined by a set of states, an initial state from this set and a set of events. We assume that starting from a given state and given the same inputs in two different executions, a process *can* display the same behavior and be in the same state. For ease in notation, we assume that the set of possible states of two different processes or channels are disjoint. A global system state is a set of component process states and channel states (only in the asynchronous case). With each system, there is an associated initial system state. In the initial system state, all the component processes are in their respective initial states and the channels (in case of asynchronous communication) are empty.

An execution of a system is a sequence of the form:

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \dots$$

where  $s_0$  is the initial system state of the system, each  $s_i$  is a global system state and each  $\alpha_i$  is an action. Each action is the execution of a primitive statement by a process and atomically transforms the state of the system. The set of all executions characterizes the behavior of the system. A system state is a *reachable state* for a system if it occurs in some execution of the system.

If  $s_g$  is a global system state of a system  $S$ , then  $s_g^P$  denotes the state of process  $P$  in the system state  $s_g$ .

**Definition 6.1** For an execution of the system  $S$ , the behavior of a process  $P$  is the sequence of triplets of the form  $\langle N, N_1, m \rangle$  or  $\langle N_1, N, m \rangle$  where  $N$  is the name of

process  $P$  and  $m$  is a message sent from  $P$  to the process named  $N_1$  in the former case and a message received by  $P$  from the process named  $N_1$  in the latter.

The behavior of a process is thus the way the rest of the processes view this process and is the sequence of messages exchanged between this process and the rest of the system. Typically, the behavior of a set of processes will be an infinite sequence but we shall restrict ourselves to finite prefixes of behaviors i.e. we shall only talk of behavior observed till a point in the execution, a finite time after the starting.

**Definition 6.2** A state  $s$  of a process  $P$  is *reachable after behavior  $b$*  in system  $S$  if there is a reachable system state  $s_g$  of  $S$  such that  $s_g$  can be reached after process  $P$  has exhibited behavior  $b$  and  $s_g^P = s$ .

To simplify the presentation, we first consider an on-line change to only a single process of a distributed system. Changes to more than one processes will be considered later. We denote by  $S_{P'}^P$  a system which is same as the system  $S$  except that the processes  $P$  in  $S$  is replaced by  $P'$ . (the name of this process remains the same.)

We now define an on-line change from a system  $S$  to  $S'$ .

**Definition 6.3** An on-line change from  $S$  to  $S' = S_{P'}^P$ , using state mapping  $\mathcal{S}$  from time  $t_1$  to  $t_2$  is equivalent to the following sequence of steps.

1. At time  $t_1$  (and in state  $s$ ), the process  $P$  is stopped. The rest of the processes continue to run.
2. At time  $t_2$ , the process  $P'$  is run from state  $s' = \mathcal{S}(s)$ .

It can be seen that the above definition allows only the affected process to be stopped while the change is affected. During the time that the change is effected (from  $t_1$  to  $t_2$ ), the rest of the processes keep running but if any of these processes requires to receive a message from the changed process or send a message to the changed process (only in the synchronous case), it must wait till the change is complete and the new version of the concerned process starts running. We assume that this delay

is not large enough for the waiting process to time out and assume that the other process has terminated or the processor running it has crashed and take some action (e.g. for fault tolerance) which would not be taken in the normal course of things. Thus we require that the upper bound, if any, on the message delay is larger than the time taken for change. Clearly, this calls for an efficient implementation of the on-line change system. We now define the validity of an on-line change in this model.

**Definition 6.4** An on-line change from  $S$  to  $S'$  is *valid* if in a finite amount of time after the change, the system reaches a reachable state of  $S'$ .

## 6.2 Conditions for Validity

We now develop a set of sufficient conditions for ensuring validity of an on-line change to a distributed system in the system model described above. We want these conditions to be local to the changed process i.e. the conditions should only put restrictions on the state of the changed process so that it is not necessary to stop the rest of the system to check whether they are satisfied at any point in time or not. Further the conditions should not put any restrictions on the unchanged processes, since the code of unchanged processes may not be available in a distributed system.

As in the sequential and object-oriented models, there are two possibilities for ensuring the validity of an on-line change. One is to give conditions which ensure that the state just after the change is a reachable state of the new system and the other is to allow the state after the change to be an unreachable state of the new version and to ensure that this state always leads to a reachable state. In the distributed system case, however, the second possibility does not seem practical. This is because to ensure conditions based on “all affected variables ultimately getting redefined” one would have to analyze all the processes including the unchanged ones in conjunction because even if the state of only the changed process is “corrupted”, the whole system state is likely to get “corrupted” a short time afterwards due to inter-process communication. Further, it will also be much harder to ensure that the program does not crash because of being executed from an unreachable state. For example, it is very difficult for a process to find out whether the non-arrival of an expected

message is due to this corruption or due to processor or link failure. Because of these reasons, we concentrate on conditions that ensure that the state of the system just after the change is a reachable system state for the new version of the system.

It is clear that the state  $s'$  of  $P'$  being a “valid” state for  $P'$  (i.e. part of some reachable system state of  $S'$ ) is not sufficient for validity since validity requires the total system state to be a reachable state for the new version of the system. Thus we also require the state  $s'$  to be “consistent” with the rest of the system state just after the completion of change. This can be achieved if the state  $s'$  is such that it could have been reached in  $P'$  after the same behavior as was shown till the time of change by process  $P$ . This is because only the behavior of a process and not its internal state is visible to the rest of the processes. Thus if  $s'$  could have been reached after the same behavior as the one actually shown by  $P$  till the time of change, the rest of the processes are not aware of the change. The following theorem formalizes this idea.

**Theorem 6.1** *An on-line change from  $S$  to  $S' = S'_{P'}$  using state mapping  $\mathcal{S}$  at a time when the state of  $P$  is  $s$ , is valid if the state  $s' = \mathcal{S}(s)$  is reachable by process  $P'$  after behavior  $b$  in system  $S'$  where  $b$  is the behavior exhibited by process  $P$  till the time of change.*

**Proof:** Let  $s_g$  be the global state just after the change is initiated,  $s_{g_1}$  the global state just before the change is completed<sup>1</sup> and  $s'_{g_1}$  the global state just after the change is completed. Clearly  $s'_{g_1} = (s_{g_1} \setminus \{s\}) \cup \{s'\}$ . Let the execution of the system till just before the change is completed be called  $E$ . By assumption, there is an execution  $E'$  of the system  $S'$  during which  $P'$  shows behavior  $b$  and at the end of which  $P'$  is in state  $s'$ . Since the processes  $P$  and  $P'$  show the same behavior in executions  $E$  and  $E'$  respectively, the state of the rest of the processes at the end of these two executions *can* be the same i.e. if the system state at the end of  $E'$  is not  $s'_{g_1}$ , there is another execution  $E''$  of  $S'$  at the end of which the system state is  $s'_{g_1}$ . This implies that  $s'_{g_1}$  is a reachable state of  $S'$ . ■

---

<sup>1</sup>This state will typically be different from  $s_g$  since the unchanged processes keep executing while the change is in progress.

### 6.3 Ensuring the Sufficient Conditions

For ensuring validity using Theorem 6.1, it has to be checked that the state  $s'$  of the changed process just after the change is a part of a reachable state of  $S'$  which could have been reached after  $P'$  had exhibited behavior the same behavior  $b$  as actually shown by the process  $P$  till the time of change. This can be ascertained by showing that the state  $s'$  is part of a reachable system state of *some* system containing  $P'$  and could be reached in that system after  $P'$  has exhibited behavior  $b$ . This is because if  $s'$  can be reached in  $P'$  after behavior  $b$  in some system, then it can be reached after  $b$  in any system in which the rest of the processes (except  $P'$ ) can give inputs to  $P'$  as in  $b$ . The system  $S'$  certainly satisfies this requirement as exemplified by the execution till the time of change of the system  $S$ . Checking that  $s'$  is reachable in  $P'$  in some system containing  $P'$  can be done based only on an examination of the processes  $P$  and  $P'$  by simply treating the “receive” and “send” statements as ordinary input/output statements and then using the techniques developed in Chapter 3 for sequential programs. Treating “receive” statements as ordinary input statements removes the restrictions imposed by the sending process on the values that can be read in a “receive” statement and the analysis can therefore only ensure that the state is reachable in  $P'$  in *some* system containing it.

Thus if we consider the simple procedure-less model of Chapter 3 augmented with statements for sending and receiving messages, we can ensure reachability of  $s'$  in  $P'$  by requiring that the change be done at a control point in  $P$  which is such that all the “affected” variables will be redefined before any use. The problem now is to ensure that  $s'$  could have been reached in  $P'$  after the same behavior  $b$  as shown till the time of change by the process  $P$ . It can be easily verified that a sufficient condition for this is that no “send” or “receive” statements of  $P$  and  $P'$  are in the sets  $D_{P,P'}$  and  $D_{P',P}$  respectively<sup>2</sup> and for all control points  $C$  in  $P$  just before a “send” statement, no variable appearing in the expression representing the value to be sent is in  $\text{changed}(C) \cup \text{changed}(C')$  where  $C'$  is the corresponding control point in  $P'$  of  $C$ . Note that these conditions do not require saving the behavior of  $P$  till

---

<sup>2</sup>The sets  $D_{P,P'}$  and  $D_{P',P}$  are the analogues of the sets  $D_{\Pi,\Pi'}$  and  $D_{\Pi',\Pi}$  respectively described in Chapter 3.

the time of change which could be impractical in a real situation.

If a program model with functions and procedures is considered, the concept of functional enhancement can be used to ensure that  $s'$  is a reachable state of  $P'$  as before. To ensure that  $s'$  could have been reached after the same behavior  $b$  as shown by process  $P$  till the time of change, it is clearly sufficient to require that for a functionally enhanced function  $f$ , if execution of  $f$  from a state  $s_1$  can produce behavior  $b_1$  then the execution of the  $f'$ , the new version of  $f$  from state  $\mathcal{S}(s_1)$  can also produce the behavior  $b_1$ .

## 6.4 Changes to Multiple Processes

We now discuss some issues relating to on-line changes in which many processes are required to be changed simultaneously. Though the framework and the sufficient conditions can be easily generalized for this case, ensuring the sufficient conditions seems to be a difficult task. To extend the framework, we define a *subsystem* of a system as a subset of the set of processes of the system and the channels between these processes. For an execution of a system, the *behavior* of a subsystem of the system is defined to be the sequence of messages exchanged between the subsystem and the rest of the system. The state of a subsystem in a system state  $s_g$  is defined to be the subset of  $s_g$  comprising of the states of the processes and the channels in the subsystem. Definition 6.2 can now be naturally extended to define the reachability of a state in a subsystem of a system after a given behavior. An on-line change to processes  $P_1, P_2, \dots, P_n$  can be viewed as replacing the subsystem consisting of the processes  $P_1, P_2, \dots, P_n$  by a new version. The state mapping  $\mathcal{S}$  is assumed to map the state of this subsystem to those of its new version and the definition of an on-line change is accordingly modified. With these generalizations, Theorem 6.1 can work with a changed subsystem instead of a changed process.

However, there is considerable difficulty in ensuring the conditions of Theorem 6.1 in the case where the changed subsystem consists of more than one process. In effect we have to ensure that the state obtained by mapping the state of the changed subsystem at the time of the change is reachable for the new version of the subsystem

after the behavior shown by the changed subsystem till the time of the change. This can not be done by analyzing the changed processes individually since their concurrent execution must be considered. Thus we must somehow analyze all the changed processes “taken together”. There does not seem to be any easy way of doing this and we leave it as an open problem.

There are also implementation difficulties involved with changes to multiple processes. Suppose we are able to obtain a checkable condition on the state of the changed subsystem at the time of change which implies validity of change. The problem now is to stop the subsystem in a state which satisfies this condition. This will require a distributed protocol the nature of which will depend on the sufficient conditions developed.

There are also problems in implementing the state mapping. It may be desired to map the state of a channel between two changed processes. Thus before the state can be mapped, the state of all the channels must be captured. When the changed processes are stopped, some messages may still be in transit and therefore the channel state is not known till the all the sent messages have been queued up at the recipient process. Since this may not happen for an arbitrarily long time, a protocol is required to capture the channel state. The problem is similar to the distributed checkpointing problem and similar solutions (e.g. see [CL85]) may be used.

It may be useful to point at this stage that if in a change many processes are changed but the “semantics” of the changed processes as seen by the rest of processes are not changed or merely “enhanced” (i.e. the set of behaviors that can be exhibited by the new version of a changed process is a superset of the set of behaviors that could be shown by the older version), the change can be effected as a sequence of changes each involving a single process and thus can be easily handled. Validity of each of these on-line changes implies the validity of the overall change.

## 6.5 On-line Change for an RPC Based Program Model

As seen in the preceding section, there are considerable difficulties in ensuring validity of an on-line change in an unrestricted message passing model, especially if many processes are changed. The problem may become simpler if the message passing is restricted in some manner. In this section, we consider one such restriction namely a remote procedure call (RPC) [BN84, Bir85] model. The RPC paradigm is attractive because its semantics closely resemble those of sequential procedural languages. Since the mechanism is invariably implemented using message passing, it can clearly be modelled in the general system model presented. In the RPC model, the processes are divided into two sets, the *clients* and the *servers*. A client communicates to a server by requesting it to execute a procedure call on its behalf. The client blocks till the server has executed the requested procedure and returned the results. A single client can communicate with many servers and conversely, a server can accept requests from many clients. Note that although a client can not service any RPC requests, a server can make RPC requests to another server to fulfill an RPC request made by some client.

RPC servers are often stateless [SGK<sup>+</sup>85]. In such systems, the servers keep no state across service of RPC requests. Stateless servers are often used so that if a server crashes and then comes up again, the clients need not be restarted. Indeed, the clients need never know that the server had crashed although some requests may take longer than usual to get processed. Since such systems are easier to handle than those in which servers have state, we consider them first.

### 6.5.1 Systems with Stateless Servers

It is clear that in such a system, a client can in no way influence the behavior of another client and hence is equivalent to a system consisting only of clients with no communication between them and each having its own copy of the code of the procedures which it calls remotely. Thus the techniques developed in Chapter 3 can

be applied here to ensure validity. Let us consider the possible types of changes and see how this can be done.

Suppose, only the code of some clients changes. In this case, we analyze each client independently using techniques developed in Chapter 3 and install the change independently in each changed client. Because of the assumption of stateless servers, validity of change for each client will clearly imply the validity of change for the distributed system as a whole.

Next suppose a server changes in such a way that the new version of its changed procedure is a functional enhancement (ref. Definition 3.7) of its older version. Now if we consider it as a change in each client which calls this procedure (directly or indirectly), Theorem 3.3 tells us that having the changed functions off the stack at the time of change is sufficient to ensure validity for each client. In terms of the actual distributed system, it means that stopping the server at a time when it is not executing any of the changed functions and then installing the change is sufficient to ensure validity of change.

In the more general case, the new versions of the changed functions of the server are not functional enhancements of their new versions. Typically, in this case the code of the clients which call these functions will also change. Even if it is not the case, we have to consider the clients also as changed since the timing of change will, in general, also depend on the state of clients. Again, we consider the equivalent system consisting only of clients and consider the validity of change for each client. In general, to ensure the validity of change for each client, we will have to have some unchanged functions also off the stack at the time of change, as seen in Chapter 3. Thus we get a set of functions of the changed clients and servers which must not be executing at the time of change, in order to ensure validity. The problem is now to stop these processes at a time when this condition is true. To do this, we first stop each client whenever its required functions are off the stack. When all the changed clients have stopped in their respective desired states, we stop each server whenever its required functions are off the stack. Note that if we stop a server before we stop some clients, some client may not be able to reach a state in which all its required functions are off the stack. In fact, it may be necessary to impose an ordering even

among the servers since servers are allowed to make RPC calls to other servers. In general, one can make a call-graph of the set of changed processes in which an edge from  $P_1$  to  $P_2$  indicates that a function required to be off the stack in  $P_1$  can make an RPC request to  $P_2$  and then stop the processes in the order indicated by a topological sort of the graph. Clearly, this can be done only if the graph is acyclic. However, if the graph contains a cycle, then the system itself is prone to deadlock and is therefore not considered.

### 6.5.2 Systems with Stateful Servers

If the servers are not stateless, then we can no longer view the system as a collection of independent sequential processes since the behavior of a client may affect that of another through the state stored in the server. In this case, the model becomes as powerful as the unrestricted message passing model and as before Theorem 6.1 has to be used to ensure validity. Thus in such systems, changes to multiple processes are difficult to handle. However, in practice, the system is expected to have mostly stateless servers with only a few stateful servers to implement the necessary synchronization etc. In such a scenario, the definition of the behavior of a process need only include messages (RPC requests and replies in this context) to and from stateful servers. This can be easily seen by considering an equivalent system with no stateless servers in which the code of stateless servers has been replicated in each of its clients as before. Thus if multiple processes change but in such a way that the change affects only the processes' communication with the stateless servers, the changes can be installed one by one as explained earlier.

## 6.6 Conclusions

In this chapter, we have extended our framework for studying on-line software changes to distributed systems. Ensuring validity of an on-line change to a distributed system is more difficult than for a sequential program because although the system must be guaranteed to reach a *global* reachable state of the new version after the change, restrictions on the timing of change can only be placed in terms

of conditions on the state of the changed processes. Further it is assumed that the code of the unchanged processes is not available for analysis. The development of sufficient conditions for validity has been done on the same general lines as for sequential programs. However, unlike in the sequential case, a condition similar to “affected variables getting ultimately redefined” does not seem feasible in this case due to reasons outlined earlier. Further it has been shown that it is not sufficient for validity that the state of the changed process after the change is a reachable state of the new version of the changed process; this state must also be consistent with the rest of the system state. To ensure this second condition, the past behavior of the changed process must also be considered in addition to its state at the time of the change. This is another point of departure from the sequential case. The difficulties in ensuring validity and in implementation in the case where several processes are required to be changed at the same time have been demonstrated and the problem has been left open. Finally, we have considered an RPC based program model for distributed systems and have shown how the restriction imposed on message passing by insisting on stateless servers helps simplify the problem and make it more manageable.

At present there is considerable debate about the programming model and notation that should be adopted for programming distributed systems and a consensus opinion does not seem to be in sight. We have considered two popular models and shown how to ensure validity in these models and the difficulties therein. It should not however be taken to mean that no more satisfactory solutions are possible for other models of distributed programming. In view of this, the work presented in this chapter should be considered as exploratory rather than definitive in nature.

# Chapter 7

## Conclusions

On-line software version change is an important technique which can reduce to a considerable extent the time during which the services of a system are unavailable to the users due to software upgradation and thus improve the overall system availability at a relatively low cost. In conjunction with roll-back based methods, it can also be used to increase the reliability of the software at a relatively low cost as compared to the traditional methods. On-line software change is, however, still a relatively new and unfamiliar technology and the various theoretical and implementation issues associated with it must be considered and the problems involved therein solved satisfactorily before it can be put to widespread use.

The main theoretical issue involved in on-line change is that after the change, the system must behave in an expected and desirable manner. Thus one must give a definition of validity of an on-line change and give ways by which validity can be ensured. In this thesis, we have attempted to study this problem. For this purpose, we have given a formal framework for modelling on-line version changes and given a possible definition for validity of an on-line change within this framework. For a simple sequential program we have shown that determining validity is, in general, undecidable. The undecidability of validity in the simple program model implies undecidability in other more realistic program models and also that we can not hope to develop any general algorithm to check whether a given on-line change is valid or not. Hence, at best, some sufficient conditions can be given which when satisfied,

will ensure validity.

We have given a set of sufficient conditions for validity in the simple program model and have shown how they can be ensured using data flow analysis.

We have developed simpler sufficient conditions for validity for a more general program model (with procedures and functions) by treating functions as the units of change. These conditions specify which functions should be off the stack at the time of change to ensure validity. A system for implementing on-line changes for this model has also been developed. Experiments with the system have shown that there is only a little deterioration in the performance of a running program during the time when the on-line change is installed, and that the developed conditions can be easily used to ensure validity in most practical cases.

An object oriented program model was then considered and we showed how validity can be ensured in such a model. Finally a message passing based distributed system model was considered. For this model, sufficient conditions for validity were first given for the case where only one process is changed across versions and then it was shown how these conditions can be generalized for the multiple process change case.

One can give alternate characterizations of the notion of validity. Any such definition must, on one hand, capture the user's expectations about the program behavior after the change and on the other, should also be realistic i.e. it should be achievable in most practical situations. We feel however that ours is the weakest possible definition of validity that would be acceptable to the user of the software since at the very least, the user will want the system to start behaving like the new program version some time after the change. Thus any other definition of validity would be at least as difficult as ours to achieve.

Our framework for modelling on-line changes and their validity is based on the concept of process states. We have chosen this approach since the notion of state can be meaningfully used in a wide variety of different program models as illustrated in this thesis. One could have chosen other ways of modelling on-line changes and their validity. For example, a definition of validity based on the concept of pre and post conditions has been presented in [GJ93a]. However the applicability of such

models in limited. Moreover, the explicit use of the concept of states allows us to naturally model the mapping of state at the time of change which would be very often required in practice and indeed without which validity would not be achievable in many real situations.

Because of the inherent difficulty of the problem it is expected that no realistic system can hope to automatically guarantee validity of change in all practical situations without enlisting some aid from the user. Some of the conditions presented in this thesis are therefore meant as guidelines to the user rather than for automatic checking. Tools can however be developed to assist the user in his task of ensuring validity. For example, a suitably modified version of algorithm *Change* can be used to compare two version of a changed function for determining the set of variables which are affected differently by the two versions. The user, based on his knowledge of the program, may then be able to provide a suitable mapping for these variables and thus ensure that the new version of the function is a functional enhancement of the old one.

In this thesis, we have only considered a few relatively simple program models and with some simplifying assumptions e.g. input from files is not considered. It may be worth investigating validity of change in other models as well. For example, many popular languages (e.g. C++) combine the object-oriented and procedure based paradigms of computing. Shared memory based models of distributed computing should also be considered. To overcome the problems is ensuring validity in distributed systems, it may be worthwhile to investigate the issue in concurrent object-oriented languages.

As mentioned in Chapter 6, we leave open the problem of ensuring validity of an on-line change to a distributed program for the case when multiple processes are desired to be simultaneously changed. A solution to this problem will allow communication protocols to be updated on-line which can not be done by changing the individual processes one by one.

Experience is needed with on-line change for object-oriented and distributed programs so that the problems involved in ensuring validity in practical situations can be understood. Experiments on the lines of the one described in Chapter 4 can

go a long way in bridging this gap. As shown by these experiments, the “validation” of an on-line change is also a crucial issue. Future on-line version change systems should also include facilities for validation.

Some systems for which on-line change might be desired in practice are of a real-time nature. For such systems, the any notion of validity of change must also ensure that no real-time constraints are violated. The presented framework is incapable of handling such constraints. Future research should therefore address this question.

# References

- [AF89] Y. Artsy and R. Finkel. “Designing a process migration facility: the Charlotte experience”. *IEEE Computer*, 22(9):47–56, 1989.
- [Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.
- [AK87] P. E. Ammann and J. C. Knight. “Data diversity: an approach to software fault tolerance”. In *Proc. 17th International Conference on Fault Tolerant Computing Systems*, pages 122–126, Pittsburgh, 1987.
- [AL81] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice Hall International, 1981.
- [ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Co., 1986.
- [Avi85] A. Avizienis. “The N-version approach to fault tolerant software”. *IEEE Tran. on Software Engg.*, SE-11(12):1491–1501, Dec 1985.
- [BDW90] M. R. Barbacci, D. L. Doubleday, and C. B. Weinstock. “Application-level programming”. In *Proc. IEEE International Conference on Distributed Computing Systems*, pages 458–465, 1990.
- [Bir85] A. D. Birell. “Secure communication using remote procedure calls”. *ACM Trans. on Computer Systems*, 3(1), February 1985.
- [BJ82] D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. PHI Series in Computer Science. Prentice-Hall Inc., Eaglewood Cliffs, New Jersey, 1982.
- [Blo83] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, March 1983.
- [BN84] A. Birell and B. Nelson. “Implementing remote procedure calls”. *ACM Trans. on Computer Systems*, 2(1), February 1984.

- [Che88] D. Cheriton. “The V distributed system”. *Communications of the ACM*, pages 314–333, 1988.
- [CL85] K. M. Chandy and L. Lamport. “Distributed snapshots: determining global states of distributed systems”. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
- [Cox86] B. J. Cox. *Object Oriented Programming: an Evolutionary Approach*. Addison-Wesley Publishing Co., Reading, Mass., 1986.
- [Deo90] N. Deo. *Graph Theory with Applications to Engineering and Computer Science*. Prentice Hall of India Pvt. Ltd., 1990.
- [DG87] L. G. DeMichiel and R. P. Gabriel. “The Common Lisp Object System: An overview”. In *ECOOP '87, European Conference on Object Oriented Programming*, pages 151–170, Paris, France, June 1987. Lecture Notes in Computer Science, Vol. 276.
- [Fab74] R. S. Fabry. “Capability based addressing”. *Communications of the ACM*, 17(7):403–412, July 1974.
- [Fab76] R. S. Fabry. “How to design systems in which modules can be changed on the fly”. In *Proc. 2nd Int. Conf. Software Engg.*, 1976.
- [FHKR93] G. Fowler, Y. Huang, D. Korn, and H. C. Rao. “A user-level replicated file system”. In *Proc. Summer Usenix*, pages 279–290, June 1993.
- [FS91] O. Frieder and M. E. Segal. “On dynamically updating a computer program: from concept to prototype”. *J. System Software*, 14(2):111–128, September 1991.
- [GH78] J. V. Guttag and J. J. Horning. “The algebraic specifications of abstract data types”. *Acta Informatica*, 10:27–52, 1978.
- [GIL78] H. Goullon, R. Isle, and K. Lohr. “Dynamic restructuring in an experimental operating system”. *IEEE Trans. Software Engg.*, SE-4(4), July 1978.
- [GJ93a] D. Gupta and P. Jalote. “Increasing system availability through on-line software version change”. In *Proc. International Conference on Fault Tolerant Computing Systems*, Toulouse, France, June 1993.
- [GJ93b] D. Gupta and P. Jalote. “On-line software version change using state transfer between processes”. *Software - Practice and Experience*, 23(9):949–964, September 1993.

- [GR83] A. Goldberg and D. Robson. *Smalltalk-80, The Language and its Implementation*. Addison-Wesley Publishing Co., Reading, Mass., 1983.
- [Gra85] J. Gray. “Why computers stop and what can be done about it?”. Technical Report 85.7, Tandem Computers, Cupertino, CA, June 1985.
- [Hoa72] C. A. R. Hoare. “Proof of correctness of data representations”. *Acta Informatica*, 1:271–281, 1972.
- [HP93] C. Hofmeister and J. Purtillo. “Dynamic reconfiguration in distributed systems: adapting software modules for replacement”. In *Proc. 13th International Conference on Distributed Computing Systems*, pages 101–111, Pittsburg, May 1993.
- [HWP92] C. Hofmeister, E. White, and J. Purtillo. “Surgeon: a package for dynamically reconfigurable distributed applications”. In *Proc. IEEE International Conference on Configurable Distributed Systems*, March 1992.
- [Ing78] D. H. H. Ingalls. “The Smalltalk-76 programming system design and implementation”. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 9–16, January 1978.
- [Jal94] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, Eaglewood Cliffs, NJ, 1994.
- [KM85] J. Kramer and J. Magee. “Dynamic configuration for distributed systems”. *IEEE Trans. on Software Engg.*, SE-11(4):424–436, April 1985.
- [KM90] J. Kramer and J. Magee. “The evolving philosophers problem: dynamic change management”. *IEEE Trans. on Software Engg.*, 16(11):1293–1306, November 1990.
- [Lea91] G. T. Leavens. “Modular specification and verification of object-oriented programs”. *IEEE Software*, pages 72–80, July 1991.
- [Lee83] I. Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, University of Wisconsin, 1983.
- [Lis88] B. Liskov. “Distributed programming in Argus”. *Communications of the ACM*, pages 300–312, March 1988.
- [MKS89] J. Magee, J. Kramer, and M. Sloman. “Constructing distributed systems in Conic”. *IEEE Trans. on Software Engg.*, 15:663–675, June 1989.

- [PH91] J. Purtillo and C. Hofmeister. “Dynamic reconfiguration of distributed programs”. In *Proc. IEEE International Conference on Distributed Computing Systems*, pages 560–571, May 1991.
- [PJ89] J. Purtillo and P. Jalote. “An environment for prototyping distributed systems”. In *Proc. 9th Int. Conf. on Distributed Computing Systems*, pages 588–594, 1989.
- [PJ93] A. Parasrampurua and P. Jalote. “Exception handling in object-oriented systems”. Technical report, Dept. of Computer Science and Engg., I.I.T. Kanpur, 1993.
- [Pur90] J. Purtillo. “The polyolith software bus”. Tech. report no. 2469, Univ. of Maryland, 1990. Also to appear in *ACM Transactions on Programming Languages and Systems*.
- [Ran75] B. Randell. “System structure for software fault tolerance”. *IEEE Tran. on Software Engg.*, SE-1:220–232, June 1975.
- [Rey72] J. C. Reynolds. “definitional interpreters for higher order programming languages”. In *Proc. 25th ACM Nat’l Conf.*, pages 717–740, 1972.
- [RT90] E. S. Raymond and M. Threepoint. *A Guide to the Mazes of Menace: Guidebook for Nethack 3.0*. Thyrsus Enterprises, Malvern, PA 19355, May 1990.
- [Sch86] D. A. Schmidt. *Denotational Semantics — A Methodology for Language Development*. Allyn and Bacon Inc., 1986.
- [Seg91] M. E. Segal. “Extending dynamic program updating systems to support distributed systems that communicate via remote evaluation”. In *Proc. International Workshop on Configurable Distributed Systems*, pages 188–199, 1991.
- [SF89a] M. E. Segal and O. Frieder. “Dynamic program updating: a software maintenance technique for minimizing software downtime”. *Software Maintenance: Research and Practice*, 1(1):59–79, September 1989.
- [SF89b] M. E. Segal and O. Frieder. “Dynamically updating distributed software: supporting change in uncertain and mistrustful environments”. In *Proc. IEEE Conference on Software Maintenance*, October 1989.
- [SF93] M. E. Segal and O. Frieder. “On-the-fly program modification: systems for dynamic updating”. *IEEE Software*, pages 53–65, March 1993.

- [SGK<sup>+</sup>85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. “Design and implementation of the Sun network filesystem”. In *Proc. Summer Usenix Conference*, 1985.
- [Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.
- [Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Co., Reading, Mass., 1986.
- [Sun88] Sun Microsystems Inc., Mountain View, CA. *Sun OS reference manual*, 1988.
- [Uni80] United States Department of Defense. *Reference Manual for the Ada Programming Language*, 1980.
- [Web90] S. Webber. “The stratus architecture”. Technical Report TR-1, Stratus Computer, Inc., Marlboro, MA, March 1990.
- [WHF93] Y. Wang, Y. Huang, and K. Fuchs. “Progressive retry for software errors”. In *Proc. 23rd International Conference on Fault Tolerant Computing Systems*, pages 138–144, June 1993.
- [Wir85] N. Wirth. *Programming in Modula-2*. Springer Verlag, New York, 1985.
- [Xer81] The Xerox Learning Research Group. “The Smalltalk-80 system”. *BYTE*, 6(8):36–48, August 1981.
- [Yok90] Y. Yokote. *The Design and Implementation of Concurrent Smalltalk*. World Scientific Publishing Co., 1990.

# Appendix A

## Algorithm *Diff*

Figure A.1 gives algorithm *Diff* to compute the set  $D_{\Pi, \Pi'}$  for the given programs  $\Pi$  and  $\Pi'$ . The algorithm is recursive in nature and computes  $D_{\bar{S}, \bar{S}'}$  for sequences of statements  $\bar{S}$  and  $\bar{S}'$  from  $\Pi$  and  $\Pi'$  respectively. Running the algorithm for the complete statement sequences of  $\Pi$  and  $\Pi'$  respectively, gives the set  $D_{\Pi, \Pi'}$ . The algorithm also gives “corresponding( $S$ )” which specifies the statement in  $\Pi'$  corresponding to a statement  $S$  of  $\Pi$  not in  $D_{\Pi, \Pi'}$ .

The algorithm works by keeping two pointers  $S$  and  $S'$  within  $\bar{S}$  and  $\bar{S}'$  respectively. The statement  $S$  is matched with  $S'$  and if they are different then  $\bar{S}'$  is searched for a statement matching  $S$ . If such a statement is not found then  $S$  is added to  $D_{\bar{S}, \bar{S}'}$ . It is easy to see that the set of statements computed by the algorithm satisfies the definition of  $D_{\Pi, \Pi'}$  since it always adds a statement of  $\Pi$  to the set if the statement is not in  $\Pi'$  also. It can also be seen that it finds the minimal such set in most cases. The computed set is not minimal only if the order of some statements of  $\Pi$  is reversed in  $\Pi'$ .

It can be seen that if algorithm *Diff* is used to compute both  $D_{\Pi, \Pi'}$  and  $D_{\Pi', \Pi}$ , it is possible that statements  $S$  and  $S'$  of  $\Pi$  and  $\Pi'$  respectively are such that  $S \notin D_{\Pi, \Pi'}$ ,  $S' \notin D_{\Pi', \Pi}$ ,  $\text{corresponding}(S) = S'$  and  $\text{corresponding}(S') \neq S$ . To make sure that  $\text{corresponding}(S) = S'$  if and only if  $\text{corresponding}(S') = S$  whenever  $S \notin D_{\Pi, \Pi'}$  and  $S' \notin D_{\Pi', \Pi}$ ,  $D_{\Pi', \Pi}$  is computed simply as the set of statements  $S'$  of  $\Pi'$  such that for no statement  $S$  of  $\Pi$ ,  $\text{corresponding}(S) = S'$ . Further, for a statement  $S' \notin D_{\Pi', \Pi}$ ,

```

Algorithm Diff ( $\bar{S}, \bar{S}'$ )
begin
   $D_{\bar{S}, \bar{S}'} \leftarrow \emptyset$ ;
   $S \leftarrow$  first statement of  $\bar{S}$ ;
   $S' \leftarrow$  first statement of  $\bar{S}'$ ;
  repeat
    if  $S$  and  $S'$  are different then begin
       $S_1 \leftarrow$  search( $\bar{S}', S', S$ );
      /* Search for  $S$  in  $\bar{S}'$  starting at  $S'$  */
      if  $S_1 \neq \phi$  then
        /* found */
         $S' \leftarrow S_1$ ;
      else begin
        /* not found */
         $D_{\bar{S}, \bar{S}'} \leftarrow D_{\bar{S}, \bar{S}'} \cup \{S\}$ ;
         $S \leftarrow$  statement after  $S$  in  $\bar{S}$  ( $\phi$  if none);
      end
    end else begin
      /*  $S$  and  $S'$  are not different */
      if  $S$  is a while statement then
         $D_{\bar{S}, \bar{S}'} \leftarrow D_{\bar{S}, \bar{S}'} \cup \text{Diff}(\text{loop of } S, \text{loop of } S')$ ;
      else if  $S$  is an if-then statement then
         $D_{\bar{S}, \bar{S}'} \leftarrow D_{\bar{S}, \bar{S}'} \cup \text{Diff}(\text{then part of } S, \text{then part of } S')$ ;
      else if  $S$  is an if-then-else statement then
         $D_{\bar{S}, \bar{S}'} \leftarrow D_{\bar{S}, \bar{S}'} \cup \text{Diff}(\text{then part of } S, \text{then part of } S')$ ;
         $\cup \text{Diff}(\text{else part of } S, \text{else part of } S')$ ;
      corresponding( $S$ )  $\leftarrow S'$ ;
       $S \leftarrow$  statement after  $S$  in  $\bar{S}$  ( $\phi$  if none);
       $S' \leftarrow$  statement after  $S'$  in  $\bar{S}'$  ( $\phi$  if none);
    end
  until  $S = \phi$ 
  return  $D_{\bar{S}, \bar{S}'}$ ;
end

```

Figure A.1: Algorithm *Diff*

$\text{corresponding}(S')$  is defined to be the statement  $S$  of  $\Pi$  such that  $\text{corresponding}(S) = S'$ .

# Appendix B

## Correctness of Algorithm *Change*

Here we prove that using algorithm *Change*, we can find for a pair of equivalent control points of  $\Pi$  and  $\Pi'$  respectively the set of variables which would have had different values had  $\Pi'$  been executed in place of  $\Pi$ . We assume that in  $\Pi'$  there are no statements after any potentially infinite loop. Formally stated, the assumption is that if  $S'_1$  and  $S'_2$  are two consecutive statements of  $\Pi'$ ,  $C'_1$  and  $C'_2$  denote the control points just before  $S'_1$  and  $S'_2$  respectively and there exists a reachable state  $s'_1$  of  $\Pi'$  with  $s'_1(PC) = C'_1$  then if  $\Pi'$  is executed from state  $s'_1$ , there exists a sequence of inputs which causes statement  $S'_1$  to terminate and control to reach  $C'_2$ . Note that the assumption does not rule out infinite loops themselves. Indeed many programs for which on-line might be desired in practice are likely to be infinitely running.

The following theorem states the desired result. For simplicity, we assume that  $V(\Pi) = V(\Pi') = V$  and that  $\mathcal{V}$  is the identity mapping.

**Theorem B.1** *If  $C$  is a control point in  $\Pi$  with equivalent  $C'$  in  $\Pi'$  then for any reachable state  $s$  of  $\Pi$  with  $s(PC) = C$ , there exists a reachable state  $s'$  of  $\Pi'$  such that  $s'(PC) = C'$  and  $s(x) = s'(x)$  for each  $x \notin V_C$  where  $V_C = \text{changed}(C) \cup \text{changed}(C')$ .*

**Proof:** The result is proved by induction on the sequence of statements (starting with  $s_{\Pi_0}$ ) through which the state  $s$  is reached in program  $\Pi$ .

In the base case,  $s$  is  $s_{\Pi_0}$ ,  $C$  is the control point before the first statement of  $\Pi$

and  $C'$  is the control point before the first statement of  $\Pi'$ . In this case, for all  $x \in V$ ,  $s_{\Pi_0}(x) = s_{\Pi'_0}(x)$  and the result is trivially true.

As the induction hypothesis, we assume the result to be true for a reachable state  $s_1$  such that  $s_1(\text{PC}) = C_1$  such that the control point  $C_1$  of  $\Pi$  has the equivalent control point  $C'_1$  in  $\Pi'$ . Thus we assume that there exists a reachable state  $s'_1$  of  $\Pi'$  such that  $s'_1(\text{PC}) = C'_1$  and  $s_1(x) = s'_1(x)$  whenever  $x \in V \perp V_{C_1}$ .

In the induction step, we show the result to be true of the first state  $s_2$  reached from  $s_1$  such that  $C_2 = s_2(\text{PC})$  has an equivalent  $C'_2$  in  $\Pi'$ . Thus we show that there exists a reachable state  $s'_2$  of  $\Pi'$  such that  $s'_2(\text{PC}) = C'_2$  and  $s_2(x) = s'_2(x)$  whenever  $x \in V \perp V_{C_2}$ .

We consider below all possible cases in which  $s_2$  can be reached from  $s_1$  without encountering any control points which have equivalents in  $\Pi'$ .

$$\begin{array}{l} \text{Case 1:} \\ C_1 \perp \rightarrow S_1 \\ C_2 \perp \rightarrow S_2 \end{array} \qquad \begin{array}{l} C'_1 \perp \rightarrow S'_1 \\ C'_2 \perp \rightarrow S'_2 \end{array}$$

Here,  $S_1 \notin D_{\Pi, \Pi'}$ ,  $S'_1 \notin D_{\Pi', \Pi}$  and  $\text{corresponding}(S_1) = S'_1$ . Let  $CV_1 = \text{changed}(C_1)$  and  $CV'_1 = \text{changed}(C'_1)$ .

Case 1a:  $S_1$  is “write  $y$ ”. In this case,

$$\begin{aligned} \text{gen}(S_1) &= \text{gen}(S'_1) = \emptyset \\ \text{kill}(S_1) &= \text{kill}(S'_1) = \emptyset \end{aligned}$$

Therefore, we get  $V_{C_2} \supseteq V_{C_1}$ . Since a write statement does not define any variables, it is clear that for all  $x \in V$ ,  $s_1(x) = s_2(x)$ . Also, there exists a reachable state  $s'_2$  of  $\Pi'$  such that for all  $x \in V$ ,  $s'_1(x) = s'_2(x)$ . Since we have assumed that  $s_1(x) = s'_1(x)$  for all  $x$  in  $V \perp V_{C_1}$ , we can conclude that  $s_2(x) = s'_2(x)$  for all  $x$  in  $V \perp V_{C_2}$ .

Case 1b:  $S_1$  is “read  $y$ ”. In this case,

$$\begin{aligned} \text{gen}(S_1) &= \text{gen}(S'_1) = \emptyset \\ \text{kill}(S_1) &= \text{kill}(S'_1) = \{y\} \end{aligned}$$

---

<sup>1</sup>If  $C$  and  $C'$  are not equivalent, a dummy statement can be added to the beginning of both  $\Pi$  and  $\Pi'$ .

Therefore, we get  $V_{C_2} \supseteq V_{C_1} \perp \{y\}$ . Since  $S_1$  defines only  $y$ , it is clear that for all  $x \in V \perp \{y\}$ ,  $s_1(x) = s_2(x)$ . Also, there exists a reachable state  $s'_2$  of  $\Pi'$  such that for all  $x \in V \perp \{y\}$ ,  $s'_1(x) = s'_2(x)$  and  $s'_2(y) = s_2(y)$  ( $s'_2$  is reached from  $s'_1$  if the input  $s_2(y)$  is given.) These observations lead us to the fact that  $s_2(x) = s'_2(x)$  for all  $x$  in  $V \perp V_{C_2}$ .

Case 1c:  $S_1$  is “ $y := \text{function}(y_1, y_2, \dots, y_n)$ ”.

Case 1c(i):  $CV_1 \cap \{y_1, y_2, \dots, y_n\} = \emptyset$  and  $CV'_1 \cap \{y_1, y_2, \dots, y_n\} = \emptyset$ .

This case is similar to case 1b above.

Case 1c(ii): At least one of  $CV_1 \cap \{y_1, \dots, y_n\}$  and  $CV'_1 \cap \{y_1, \dots, y_n\}$  is not  $\emptyset$ . In this case  $V_{C_2} \supseteq V_{C_1} \cup \{y\}$ . Also, as in case 1b, for all  $x \in V \perp \{y\}$ ,  $s_1(x) = s_2(x)$ . Clearly, there also exists a reachable state  $s'_2$  of  $\Pi'$  such that for all  $x \in V \perp \{y\}$ ,  $s'_1(x) = s'_2(x)$ . These facts then imply that  $s_2(x) = s'_2(x)$  for all  $x$  in  $V \perp V_{C_2}$ .

Case 1d:  $S_1$  is “while ( $e$ ) do begin  $\bar{S}_1$ ” and  $S'_1$  is “while ( $e$ ) do begin  $\bar{S}'_1$ ”, where sequences  $\bar{S}_1$  and  $\bar{S}'_1$  are the bodies of  $S_1$  and  $S'_1$  respectively. Since  $s_2$  is reached from  $s_1$  without encountering any control points which have equivalents in  $\Pi'$ , it follows that the boolean expression  $e$  evaluates to “false” in the state  $s_1$ . This implies that  $s_2(x) = s_1(x)$  for all  $x \in V$ . Since  $S_1 \notin D_{\Pi, \Pi'}$  and  $S'_1 \notin D_{\Pi', \Pi}$ , none of the variables appearing in  $e$  are in  $V_{C_1}$ . Thus for all variables  $x$  in occurring in  $e$ ,  $s_1(x) = s'_1(x)$ . Hence  $e$  evaluates to “false” also in  $s'_1$ . Clearly therefore, there exists a reachable state  $s'_2$  of  $\Pi'$  with  $s_2(PC) = C'_2$  such that for all  $x \in V$ ,  $s_2(x) = s'_2(x)$ . Also clearly, since  $S_1$  and  $S'_1$  are while-do statements,  $V_{C_2} \supseteq V_{C_1}$ . It follows that  $s_2(x) = s'_2(x)$  for all  $x$  in  $V \perp V_{C_2}$ .

Case 1e:  $S_1$  is “if ( $e$ ) then begin  $\bar{S}_1$ ” and  $S'_1$  is “if ( $e$ ) then begin  $\bar{S}'_1$ ”. The proof is similar to case 1d.

Case 2:	$C_1 \perp \rightarrow$ while ( $e$ ) do begin $C_2 \perp \rightarrow$ $\dots$ $\dots$ $\vdots$ $\dots$ end	$C'_1 \perp \rightarrow$ while ( $e$ ) do begin $C'_2 \perp \rightarrow$ $\dots$ $\dots$ $\vdots$ $\dots$ end
---------	-------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------

Again, since  $s_2$  is reached from  $s_1$  without encountering any control points which have equivalents in  $\Pi'$ , it follows that the boolean expression  $e$  evaluates to “true” in the state  $s_1$ . Now, since  $C_2$  has an equivalent ( $C'_2$ ) in  $\Pi'$ ,  $V_{C_1} \cap \text{vars}(e) = \emptyset$  where  $\text{vars}(e)$  denotes the set of variables occurring in  $e$ . This implies that for all  $x \in \text{vars}(e)$ ,  $s_1(x) = s'_1(x)$ . This means that  $e$  evaluates to “true” in  $s'_1$  also. Also, we note that  $V_{C_2} \supseteq V_{C_1}$  and for all  $x \in V$ ,  $s_2(x) = s_1(x)$ . Clearly, since there exists a reachable state  $s'_2$  of  $\Pi'$  with  $s'_2(\text{PC}) = C'_2$  such that for all  $x \in V$ ,  $s'_2(x) = s'_1(x)$ , it can be inferred that  $s'_2(x) = s_2(x)$  whenever  $x \in V \perp V_{C_2}$ .

Case 3:	$C_1 \perp \rightarrow$ if ( $e$ ) then begin $C_2 \perp \rightarrow$ $\dots$ $\dots$ $\vdots$ $\dots$ end	$C'_1 \perp \rightarrow$ if ( $e$ ) then begin $C'_2 \perp \rightarrow$ $\dots$ $\dots$ $\vdots$ $\dots$ end
---------	------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------

The proof is similar to case 2.

Case 4:	$C_1 \perp \rightarrow$ if ( $e$ ) then begin $\dots$ $\vdots$ $\dots$ end else begin $C_2 \perp \rightarrow$ $\dots$ $\dots$ $\vdots$ $\dots$ end	$C'_1 \perp \rightarrow$ if ( $e$ ) then begin $\dots$ $\vdots$ $\dots$ end else begin $C'_2 \perp \rightarrow$ $\dots$ $\dots$ $\vdots$ $\dots$ end
---------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

The proof is similar to case 2.

Case 5:	$C_2 \perp \rightarrow$ while ( $e$ ) do begin $\dots$ $\vdots$ $\dots$ $C_1 \perp \rightarrow$ end	$C'_2 \perp \rightarrow$ while ( $e$ ) do begin $\dots$ $\vdots$ $\dots$ $C'_1 \perp \rightarrow$ end
---------	-----------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------

Here, again,  $V_{C_2} \supseteq V_{C_1}$  and since no control points with equivalents in  $\Pi'$  are encountered from  $s_1$  to  $s_2$ , for all  $x \in V$ ,  $s_1(x) = s_2(x)$ . Also there exists a reachable state  $s'_2$  of  $\Pi'$  with  $s'_2(\text{PC}) = C'_2$  such that  $s'_2(x) = s'_1(x)$  for all  $x \in V$ . Clearly, then,  $s'_2(x) = s'_1(x)$  whenever  $x \in V \perp V_{C_2}$ .

Case 6:	if ( $e$ ) then begin $\dots$ $\vdots$ $\dots$ $C_1 \perp \rightarrow$ end $C_2 \perp \rightarrow$ $\dots$	if ( $e$ ) then begin $\dots$ $\vdots$ $\dots$ $C'_1 \perp \rightarrow$ end $C'_2 \perp \rightarrow$ $\dots$
---------	------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------

Again  $V_{C_2} \supseteq V_{C_1}$ . The argument is as in case 5.

Case 7:	if ( $e$ ) then begin ... : ... end else begin ... : ... $C_1 \perp \rightarrow$ ... end $C_2 \perp \rightarrow$ ...	if ( $e$ ) then begin ... : ... end else begin ... : ... $C'_1 \perp \rightarrow$ ... end $C'_2 \perp \rightarrow$ ...
---------	----------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------

The proof is similar to case 6.

Case 8:	$S_1$ $C_1 \perp \rightarrow$ ... ... : ... $C_2 \perp \rightarrow$ $S_2$	$S'_1$ $C'_1 \perp \rightarrow$ ... ... : ... $C'_2 \perp \rightarrow$ $S'_2$
---------	------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------

Here,  $S_1, S_2 \notin D_{\Pi, \Pi'}$ ,  $S'_1, S'_2 \notin D_{\Pi', \Pi}$ ,  $\text{corresponding}(S_1) = S'_1$  and  $\text{corresponding}(S_2) = S'_2$ . Also, for each statement  $S$  between  $S_1$  and  $S_2$ , either  $S \in D_{\Pi, \Pi'}$  or  $\text{corresponding}(S) \in D_{\Pi', \Pi}$ . Similarly, for each statement  $S'$  between  $S'_1$  and  $S'_2$ , either  $S' \in D_{\Pi', \Pi}$  or  $\text{corresponding}(S') \in D_{\Pi, \Pi'}$ . Let  $\bar{S}$  and  $\bar{S}'$  denote the sequence of statements between  $S_1, S_2$  and  $S'_1$  and  $S'_2$  respectively. It is easy to see that  $V_{C_2} \supseteq V_{C_1} \cup \text{modify}(\bar{S}) \cup \text{modify}(\bar{S}')$  where  $\text{modify}(\bar{S})$  denotes the set of variables on the left side of an assignment or in a read statement, in any statement in  $\bar{S}$ . Let  $\Pi'$  be executed from  $s'_1$  giving some inputs till control reaches  $C'_2$  (such inputs do exist because of our assumption.) Let  $s'_2$  be the state obtained. Clearly, since  $s_1$  and  $s_2$  differ at most in values of variables in  $\text{modify}(\bar{S})$  and  $s'_1$  and  $s'_2$  at most in values of variables in  $\text{modify}(\bar{S}')$ ,  $s_2$  and  $s'_2$  can differ at most in values of variables in  $V_{C_2}$ .

$$\begin{array}{l}
\text{Case 9:} \\
\begin{array}{ccc}
& S_1 & S'_1 \\
C_1 \perp \rightarrow & & C'_1 \perp \rightarrow \\
& S_2 & \dots \\
C_2 \perp \rightarrow & & \dots \\
& & \vdots \\
& & \dots \\
& & C'_3 \perp \rightarrow \\
& & S'_2 \\
& & C'_2 \perp \rightarrow
\end{array}
\end{array}$$

Again,  $S_1, S_2 \notin D_{\Pi, \Pi'}$ ,  $S'_1, S'_2 \notin D_{\Pi', \Pi}$ ,  $\text{corresponding}(S_1) = S'_1$  and  $\text{corresponding}(S_2) = S'_2$ . Also, for each statement  $S'$  between  $S'_1$  and  $S'_2$ ,  $S' \in D_{\Pi', \Pi}$ . Let  $\bar{S}'$  denote the sequence of statements between  $S'_1$  and  $S'_2$ . Let  $\Pi'$  be executed from  $s'_1$  giving some inputs till control reaches  $C'_3$  (such inputs do exist because of our assumption.) Let  $s'_3$  be the state obtained. Define  $V'_{C'_1} = CV_1 \cup \text{changed}(C'_3)$  where  $CV_1$  is defined as earlier as  $\text{changed}(C_1)$ . It is easy to see that  $V'_{C'_1} \supseteq V_{C_1} \cup \text{modify}(\bar{S}')$  and  $s'_1(x) = s'_3(x)$  whenever  $x \notin \text{modify}(\bar{S}')$ . Using  $C'_3$ ,  $s'_3$  and  $V'_{C'_1}$  in place of  $C'_1$ ,  $s'_1$  and  $V_{C_1}$  respectively, the rest of the argument is similar to case 1.

$$\begin{array}{l}
\text{Case 10:} \\
\begin{array}{ccc}
& S_1 & S'_1 \\
C_1 \perp \rightarrow & & C'_1 = C'_2 \perp \rightarrow \\
& \dots & S'_2 \\
& \dots & \\
& \vdots & \\
& \dots & \\
C_2 \perp \rightarrow & & \\
& S_2 &
\end{array}
\end{array}$$

Again,  $S_1, S_2 \notin D_{\Pi, \Pi'}$ ,  $S'_1, S'_2 \notin D_{\Pi', \Pi}$ ,  $\text{corresponding}(S_1) = S'_1$  and  $\text{corresponding}(S_2) = S'_2$ . Also, for each statement  $S$  between  $S_1$  and  $S_2$ ,  $S \in D_{\Pi, \Pi'}$ . Let  $\bar{S}$  denote the sequence of statements between  $S_1$  and  $S_2$ . It is easy to see that  $V_{C_2} \supseteq V_{C_1} \cup \text{modify}(\bar{S})$  and  $s_2(x) = s_1(x)$  whenever  $x \notin \text{modify}(\bar{S})$ . With  $s'_2 = s'_1$ , it is easy to see that the result holds.  $\blacksquare$