Software Hot Swapping

Pradeep Tumati

Department of Computer Science and Applications

Virginia Tech

Thesis submitted to the faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Dr. John A.N. Lee
Dr. James D. Arthur
Dr. Stephen Edwards

January 7th 2003

Blacksburg, Virginia

Keywords: Software Hot Swapping, Dynamic Software Update, Dynamic Code Change

Software Hot Swapping

Pradeep Tumati

(ABSTRACT)

The emergence of the Internet has sparked a tremendous explosion in the special class of systems called *mission critical systems*. These systems are so vital to their intended tasks that they must operate continuously. Two problems affect them: unplanned, and therefore disastrous, downtime and planned downtime for software maintenance. As the pressure to keep these systems operating continuously increases, scheduling downtime becomes complex. However, dynamically modifying the mission critical systems without disruption can reduce the need for a planned downtime.

Every executing process has an executing code tightly coupled with an associated state, which continuously changes as the code executes. A dynamic modification at this juncture involves modifying the executable code and the state present within the binary image of the associated process. An ill-timed modification can create runtime incompatibilities that are hard to rectify and eventually cause a system crash. The purpose of the research in this thesis is to examine the causes for incompatibilities and propose the design of a dynamic modification technique: *Software Hot Swapping*. To achieve these objectives, the researcher proposes mechanisms which these incompatibilities can prevent, examines the characteristics and the implementation issues of such mechanisms, and demonstrates dynamic modification with a simple prototype *Hot Swapping* program.

Acknowledgements

I thank Dr. John A.N. Lee for his guidance and support in making this thesis possible. Without his support and his insights into numerous problems, this thesis would not have seen the light of the day. He has provided entrepreneurial encouragement by showing me how to make this technology into a marketable product. He also helped me in securing the complete intellectual rights for this research from Virginia Tech. I am grateful for all his help. I thank Dr. James D. Arthur and Dr. Stephen Edwards for serving as members of my thesis committee and for providing useful and insightful advice.

I am grateful to Shekar Nair, Vice President of AllegroNetworks, for giving me insights into the nature of the commercial mission critical systems, to Jeff Dyer and Suresh Kadiyala for their suggestions and comments on my research and the commercial potential for this technology, and to Rob Chandra, General Partner at Bessemer Venture Partners, for taking an interest in this idea and introducing me to Prabhat Goyal. I thank Prabhat Goyal for joining me in taking this technology to the next level. I also thank Dr. Jeff Ullman for making me realize that this technology has a scary commercial side. This has awakened in me a passion for solving the associated problems.

I am also indebted to my parents and the rest of my family for all the support they provided me. Last, but not the least, I thank all my friends and all others who have directly or indirectly helped me in my efforts.

Table of Contents

List of Figures

Chapter 1: Dynamic Modification of Functionality

1. Introduction

The emergence of the Internet has sparked a tremendous explosion in the special class of systems called *mission critical systems*. These systems are so vital to their intended tasks that they must operate continuously. There are two problems that can affect them: unplanned, and therefore disastrous, downtime and planned downtime for software maintenance. Although downtime can result from both hardware and software failures, this research concentrates on attempts to avoid the latter and, in doing so, focuses on *dynamic modification*, a relatively new technique that has the potential to eradicate software downtime by modifying the functionality of executing software without having to halt its execution. Accordingly, the main goals of this thesis are to examine the characteristics of dynamic modification techniques, evaluate the underlying implementation issues, and develop from the concept a flexible technology — *Software Hot Swapping* — that conforms to the typical vendor-client scenario.

Wireless applications, web servers, application service providers, and e-Commerce applications face the same problem: most modern vendors provide software systems that are critical for the continuous operation of their intended missions and any sort of downtime, planned or not, can result in client dissatisfaction. A network provider, for example, loses both revenue and the goodwill of clients if the network's service controllers are temporarily unavailable. As a result, scheduling downtime for the simple, but necessary task of updating software becomes problematic, even for such companies as Amazon.com, uBid.com, and eBay.com. Since they cannot afford to interrupt their services in order to update their software, they must use expensive server farms to avoid downtime. It is also undesirable to interrupt the execution of a network router for the sake of software maintenance. Vendors cannot predict unplanned downtime, but they can schedule a maintenance period or planned downtime. Although hot swapping can reduce a certain portion of unplanned downtime in some situations, this research concentrates on how hot swapping can eliminate the need for planned downtime.

In general, tested hardware is more reliable than software, because software must undergo versioning and maintenance more often than hardware. In the case of mission critical applications, planned downtime for software versioning is difficult to schedule — and unplanned downtime is certainly not desirable. For example, scheduling downtime for updating the software of a radar system becomes problematic. When a new service or security threat makes changes unavoidable, the changes are supposed to be designed to minimize downtime, thereby reducing system maintenance to repetitious patching.

For any mission critical system, rapid introduction of new functionality and dynamic adaptation to volatile needs are essential. An effective method has to allow both continuous operation and continuous change through dynamically modifying the functionality of the system. This means it must be possible to add a new code to a program while it is running.

Here, the term 'functionality' refers to characteristics of the program element the programmer intentionally assigns during the initial design phase. A program loaded into the memory for execution is called a process. Every executing process incorporates two components: the *executable code* and an *associated data state*. The executable code generally remains static during the process execution, while the status of the state depends upon the address associated with the instruction counter within the domain of the executable code, the state history, and inputs given to that process during the execution.

The programmer, with the help of the compiler, implements the functionality within the executable code portion of the address space of the process. Modifying the functionality of a non-executing program in secondary storage can produce a *static modification*. Dynamically modifying parts of the address space of an executing process can produce a *dynamic modification*. For example, Figure 1.1 shows a simple version change. A dynamic modification of this example entails changing the appropriate add operand present within the address space with a corresponding sub operand.

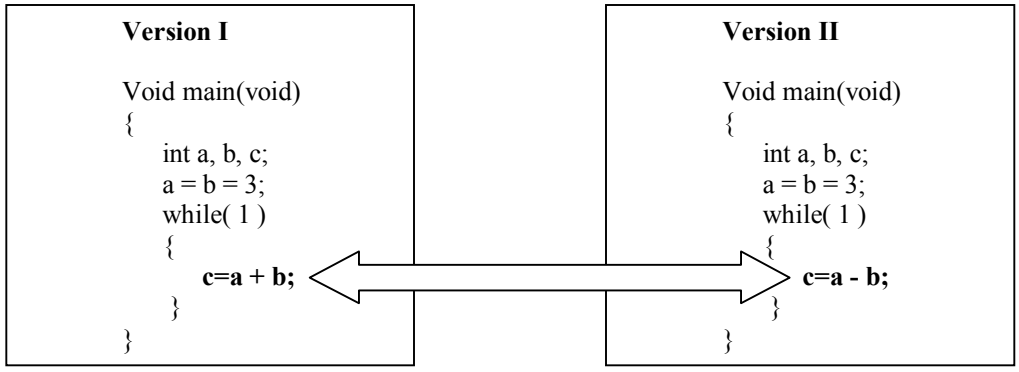


Figure 1.1: A Simple Version Change

However, in complex scenarios, if the functionality of an executing process is instantaneously modified, there is no guarantee that the existing state of the process will be *compatible* with the new functionality just created within the address space. Ensuring this compatibility makes a dynamic modification *valid*. Guaranteeing the validity is a requirement that must be met during dynamic modification. Because a non-executing program has no associated state, static modification is much simpler than dynamic modification. Gupta, Jalote and Barua describe the validity concept [OUS 6]. Gupta's doctoral dissertation further explores the validity concept [OUS 7].

To be effective, a typical dynamic modification technique must ensure the validity of dynamic modification instances. There are two approaches to solving the validity problem. The first is to detect and avoid scenarios where validity is not guaranteed. The second approach consists of controlling and modifying the state to ensure its compatibility with the newly inserted functionality. A dynamic modification technique has to use both these approaches to solve the validity problems. A system that can perform complex dynamic modifications has to have certain techniques and procedures incorporated in it that make the dynamic modifications feasible and more manageable.

The research in this thesis investigates the techniques and procedures required for performing dynamic modification.

Objectives

The objectives of this thesis are:

1. An examination of the issue of incompatibility that can arise when dynamic modification is performed at certain improper times.

2. A more complete definition of the characteristics of a typical dynamic modification technique. Segal and Frieder have already suggested some characteristics [OUS 1], but these are insufficient. Although Hicks describes some of the characteristics in his doctoral dissertation [OUS 9], because he focuses on scientific issues rather than key implementation issues that make a dynamic modification technique widely applicable, his explanation is also inadequate. Since this research studies some of the same issues as Hicks' research, it does restate some of the characteristics he identifies, but also introduces many new ones as well.

3. The identification of the general criteria and implementation issues that software engineers must address when designing a dynamic modification technique so as to determine its complexity and flexibility. Most early studies do not seem to address these topics.

4. The design of a dynamic modification technique, in this case, Software Hot Swapping, that readily adapts to any typical vendor-client scenario. This technique is designed for use in a single-threaded executing process developed using an imperative language. Adapting Software Hot Swapping to the object-oriented paradigm or to a multi-threaded scenario remains as a future goal.

5. A demonstration of a small portion of the design described in objective 4 above.

This thesis is organized as follows:

- Chapter 2 provides a description of prior work relevant to dynamic modification, and provides other information essential to understanding the concept of dynamic modification.

- Chapters 3 through 6 provide an integrated discussion of the concept, as well as underlying issues, such as potential incompatibilities. These chapters meet objective 1.

- Chapter 7, which concentrates on describing the basic characteristics and the underlying implementation issues of typical dynamic modification techniques, meets objectives 2 and 3.

- Chapters 8 through 12, which provide a detailed description of the technique of Software Hot Swapping, meet objectives 4 and 5.

- Chapter 13 provides a summary of the contributions this research makes and highlights future research projects.

Chapter 2: Literature Survey

2. Introduction

Although dynamic modification is not a new technology, it is an *ads-hoc* technique that has been in use for the past decade. Much of the previous research into dynamic modification techniques concentrates on solving issues specific to the domains for which the techniques are designed. As a result, no research provides a definition of a generalized dynamic modification approach. To date, only Gupta has attempted to reach such a formal definition in his work [OUS 6,7].

Dynamic modification is the result of several research attempts to develop mission critical systems that would not require downtime for the sake of routine maintenance. Since the number of mission critical applications is comparatively small, all such research concentrates on solving a very specific problem in each one of the existing systems. This narrow focus has resulted in research that is situation specific rather than in research that takes generalized approaches to dynamic modification. The research in this thesis, therefore, concentrates on devising a generalized approach that is also flexible.

Section 2.1 provides a survey of research to date. Section 2.2 reports on the drawbacks of the research in these studies. Finally, Section 2.3 gives a description of the motivation behind this research.

2.1 Previous Work

Although using the hardware approach or the software-based dynamic modification approach can achieve reductions in downtime, the main intention of the research in this thesis is to examine the issues a software-based approach raises. However, before looking at software-based dynamic modification systems, it is valuable to examine the hardware-based approach briefly in order to describe alternative approaches and highlight their drawbacks.

To perform dynamic modification on a program using the hardware approach, it is necessary to stop a computer with the old version of the program executing at a safe point, and simultaneously start a different computer with a new version of the program executing within it. Although the update procedure can cause the loss of some work in progress, the time required to perform this kind of modification can be significantly less, because it is possible to make a slight modification in such a way that the old system and the new system execute concurrently for a certain period of time known as the *transition period*. During this phase, old requests are left to execute while new requests are directed to the new system. After the completion of the transition, the system administrators can turn off the old system, as it will no longer serve requests. The principle disadvantage of this method, which is typically used in systems that need redundant hardware anyway to provide fault-tolerance, is its substantial cost. Also, this method is neither flexible nor generalized. Rey describes a hardware-based system used at Bell systems that can be implemented in the form of a software-based approach as well [EX 11]. With this

method, also known as the *client-server approach*, a process typically executes within the server and, by using client-server techniques, all of the clients can use the functionality of that process. Whenever the functionality must be modified within the server, a new process begins at the server, and all new client requests are routed to the new process with the old client requests left to the old process. Once the old process completes all remaining or old requests, it is terminated. This approach is currently being implemented within one of products of Sun Microsystems [EX 20]. As the concerned information is proprietary, specific details of that project are not available. Drawbacks to this approach are that it is neither flexible nor generalized.

Sections 2.1.1 through 2.1.12 report on literature devoted to all of the twelve related software-based dynamic modification systems. Most of the modern processors try to keep the data and the executable code as close to the processor as possible within the memory hierarchy. As cache memory is the closest, in terms of run-time efficiency, there are two possible places where copies of the executable code might exist: the primary memory and the cache memory. Section 2.1.13 looks at this cache coherence problem in some detail. The operating system also plays a key role in the execution of the code. For example, since executable code is stored in write-protected pages, it is possible for the operating system to block attempts to modify it. Section 2.1.14 provides a discussion of this issue. Section 2.1.15 takes into consideration the difference between *process halting* and *process suspension*. Finally, Section 2.1.16 reports on the address space structure of a running program.

2.1.1 Dynamic Type Replacement System

Fabry describes a system in which the programmer can implement or change abstract data types or modules “on the fly” [OUS 15]. Several processes can access the same module, while a module can manage permanent data, local to one process or shared by several processes. However, Fabry only considers the case in which the interface and the semantics of the module being replaced do not change across versions. This system relies on *capability-based addressing* [OUS 16]. Three specific ideas lie behind such addressing: a capability is a special kind of address for a module, only the system can create such addresses, and, in order to use any module, it is necessary to access it via one of the addresses [OUS 16]. The advantage of using a capability as an address is that its interpretation is context independent; in other words, it supplies a module with an absolute address. It is possible to specify the capability of the module within the call instruction by pointing to an indirect word containing directions to the new module’s address.

There are several drawbacks to the dynamic type replacement system. Changing the implementation of an abstract data type can require restructuring the permanent data it manages. Under Fabry’s system, this is done not at the time when the change is installed but after the change, when an instance of the data structure is first used. Since restructuring all instances of a data type can be time-consuming, this development leads to a significant reduction in the speed of the change. In addition, version numbers are used to detect obsolete data. Thus, the entry code for a module “checks” if the version

number stored in the data structure is less than the version number of the code; if this is the case, the programmer initially develops a conversion routine, which is then used whenever a conversion is required to accomplish the necessary restructuring. Since such restructuring occurs on demand, it is possible for an instance of a data type to be several versions out-of-date at its first use. To alleviate this problem, each conversion routine can call into play the previous one.

2.1.2 DAS

The experimental Dynamically Alterable System (DAS) dynamically updates programs by allowing the replacement of a module by a new version having the same interface [OUS 13]. The design of the system makes dynamic modifications possible in all system components, except in the kernel that contains part of the modification facility, which is conceptually similar to Fabry's system. *Replugging*, a mechanism based on the addressing scheme of the system implements dynamic updating. The workings of the mechanism are simple. 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. An address space transition performed by a special `call` instruction that replaces some segments in the virtual address space achieves a call to a procedure of a different module. Changing the links in the descriptor chain (the linked list described above) that point to the newly modified module results in replugging, and data restructuring is done on request rather than on demand. Again, since multiple processes are allowed to access the same modules, a locking scheme synchronizes a module's users and the replugging mechanism.

The initial design of the software architecture of DAS allows "on the fly," future modifications without the sorts of system failures ill-timed updates usually cause. (Chapter 4 provides an in-depth discussion of such problems.) The main problem with this approach, however, is that the programmer has to design the software architecture in a very special way, because upgrading must occur at predetermined times, a concept further explained in Chapter 5, Section 5.1.

2.1.3 Argus

Argus is an integrated, CLU-based programming language and system designed to support construction of distributed software and supply support for atomic transactions and crash recovery [OUS 3, 10]. An Argus program consists of a set of servers called *guardians*, which are similar in nature to an abstract data type. A remote procedure call-like mechanism with a specially designed dynamic module replacement facility accomplishes communication between guardians. 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 generate only those event sequences that the replaced abstraction in the state in which the replacement occurred permits. This condition gives the required relationship between the abstract specifications of the new and old subsystem versions is given, as well as the restrictions on the state in which the change takes place.

The main weakness of Argus is that it demands closely related behavior between the new version of a replaced subsystem and the old one. In general, dynamic modification implies type as well as code modifications, but Argus can perform only the latter. Moreover, since it is sometimes impossible to replace a subsystem by re-implementing the same abstract specifications, another drawback of the replacement system is that it requires the Argus crash recovery facilities in order to work properly and, therefore, cannot always be adaptable to other systems. Furthermore, a subsystem can be too large a unit of change for some applications.

2.1.4 Conic

Conic is a system developed at the Imperial College, London for supporting dynamic reconfiguration of programs [OUS 11]. It provides a language and a run-time environment for constructing distributed programs. A program in Conic consists of a set of modules, each of which has a number of entry and exit ports, which serve as unidirectional links and through which modules communicate with each other. A language permitting the entry port of one module to link with the exit ports of another specifies system configuration separately. Thus, communication is based on the naming of ports.

In Conic, a configuration change specification made during execution impacts the entire system. This specification can ask for the creation of new instances of modules, the deletion of old instances, or the creation and deletion of links between ports of various modules. The configuration manager translates such requests.

Conic language ensures that connections are always well typed. The problem with Conic is that although it can redirect connections between processes, it does not provide system-level support to update the code or a running process, as is possible in Argus. As a result, the programmer must code Conic applications to capture and transfer their state, which is similar to but not as flexible as the application-level approach proposed for Argus [EX 10].

2.1.5 Durra

Barbacci, Doubleday, and Weinstock have designed the language Durra to support the development of distributed applications on heterogeneous machine networks [OUS 17]. Like Conic, Durra specifies the behavior of the system components separately from that of the system structure. It also provides support for the run-time reconfiguration of the system/usage facilities 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. The consequence of this is that, in Durra, the programmer must anticipate and specify all reconfigurations in advance. This requirement causes significant programming overhead.

2.1.6 DMERT Operating System

AT&T's 3B20D processor, a part of the Number 5 Electronic Switching System, runs the Duplex Multiple Environment Real-Time (DMERT) operating system. The Field Update Subsystem of DMERT updates and installs emergency patches into the C functions of the 3B20D's switching software.

DMERT supports dynamic updating by processing a transfer vector that provides a level of indirection between a function call and the actual address of the function in memory [OUS 12]. Changing the address for a particular function in the transfer vector ensures that future references to that function are routed to the new version. The Field Update Subsystem automatically enacts such changes and updates the disk-based program images and logs of the programs running on the 3B20D as well.

The disadvantages of DMERT OS are that it assumes that the interfaces of the functions do not change between versions and that it is an application dependent technique.

2.1.7 Dynamic Linking in Operating Systems

In an operating system, dynamic linking binds properties to their corresponding identifier whenever the executable code present within the executing process references that identifier during execution. Shared libraries provide a good example [SL 1-6, SL 9-13]. The main drawback to most of these methods is that they can link modules but not unlink them. Thus far, the only technology that can do both is the "*dld*" dynamic loader, designed as part of the GNU project to dynamically load and unload routines into a running program [SL 4]. Unlinking takes two forms: soft and hard. Soft unlinking means that it is possible to reclaim memory occupied by a particular module only when the reference count of that module is zero. In contrast, with hard unlinking it is possible to unlink a module and reclaim its memory, even though there are some references to that module.

The main disadvantage of this method is that it does not perform type checking before linking. It merely verifies the identifier name and links the appropriate modules to it. Without type checking, these methods can be considered as mere formal ways of binding identifiers to a program.

2.1.8 Dynamic Classes Concept

Dorward, Sethi, and Shopiro have developed the concept of dynamically loading a class whenever invoked at the AT&T laboratories [SL 16]. This concept extends the mechanism of dynamic linking to preserve the type safety of C++ and to work at the class level. This method permits the dynamic loading of a newly-derived class of a known base class into a program. First, the programmer initiates a dynamic linker that links the shared library, which contains the implementation of the new class, into the program's address space. Then a standard factory mechanism calls the library and creates an instance of the new class [SE 8]. The instance is then cast to the type of the base class and can be used

wherever it is required. Since all calls to the base class are type-checked statically, and the base class constrains the derived class to have the same function signatures, type safety is preserved.

The main drawback of this method is that the class cannot then be unloaded, a problem that eventually led to the development of the dynamic C++ class [SL 7], which extends the concept by (1) allowing replacement of a previously loaded class with a new version and (2) allowing multiple versions (instances) of a class to coexist. Whenever a class needs to be instantiated, the underlying system uses the latest available version of the associated class. Hence, old instantiations can co-exist with new instantiations. The underlying system can also introduce new classes, provided they use a known interface.

A similar approach involves the dynamic loading of classes in Java using the Java Class Loader mechanism [SL 8]. It enables the Java Virtual Machine to load classes without knowing anything about the underlying file system semantics, and applications to dynamically load Java classes as extension modules. Hence, whenever a class needs to be instantiated, the corresponding class loader loads the latest version of the associated class.

Although this technique is similar to the idioms Coplien proposes [SL 17], there are several disadvantages to this methodology. The first major disadvantage is its suitability only for course-grained classes. When used for fine-grained classes, the associated runtime overhead increases as much as 5000% in the case of C++ [SL 7]. The second is its need for an extra discipline in coding that potentially reduces run-time performance, because existing legacy programs cannot use dynamic classes without modifying their structure. The third arises from the restriction on how dynamic classes are inherited. Finally, the fourth disadvantage is that the behavior of static methods in a dynamic class interface is not defined.

2.1.9 DYMOS

In his doctoral dissertation, Insup Lee presents a dynamic modification system called DYMOS, which is a fully integrated environment for software development and program updating OUS 2]. DYMOS, which allows individual procedures of a program to be changed, supports programs written in StarMod, a concurrent language that 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 integrated nature of the system makes the source code, the object code, and the symbol table available at all times to aid in on-line changes, and the system will ensure that the program image in its memory corresponds to the latest version of the source code.

Lee also gives a procedure for partitioning a change into a sequence of smaller changes. This decomposition is done in such a way that the intermediate programs obtained after each smaller change should be “functionally consistent.” This method has a few drawbacks, all of which Section 2.2 reports on in detail.

2.1.10 PODUS

The Procedure-Oriented Dynamic Updating System (PODUS) is an in-line software version change system developed at the University of Michigan and later enhanced at Bellcore [OUS 1, 5]. As its name suggests, PODUS supports dynamic modification to procedural programs. 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*, the user-interface to the system, provides commands for loading, running, and dynamically updating programs and then sends these requests to the *pup,* in whose address space the user program is run. Moreover, the *ush* and the *pup* communicate using Internet domain sockets and, therefore, need not reside on the same physical computer during the dynamic modification process.

PODUS is based on a *large sparse address space* architectural model, in which the address space is partitioned into a number of version spaces using a "version id" in the address. In addition to a binding table, each version space holds the program's data and code.

PODUS also supports dynamic changes to distributed programs in which processes communicate through the Remote Procedure Communication (RPC) mechanism. An enhanced version, called REV PODUS, also supports the remote evaluation mechanism of inter-process communication.

In PODUS, the *pup* updates a procedure only when it is inactive. A procedure is defined as inactive at a given time if it does not have a stack frame and none of the procedures that its new version can directly or indirectly call have stack frames in the runtime stack. These conditions, which imply that at no time can the new version of a procedure call the old version of another procedure, resemble Lee's conditions for partitioning changes into a sequence of smaller ones.

However, PODUS also allows the specification of semantic dependencies between procedures, thereby permitting the update of two or more procedures simultaneously. Since it is impossible to infer these dependencies from the syntax of the program, the user must specify them. One disadvantage is that no guidelines are given for determining them. This method has a few other drawbacks, which Section 2.2 reports on in more detail.

2.1.11 Gupta's Work

Gupta has developed a framework for performing dynamic update as a state transfer between programs written in C [OUS 6, 7]. This framework permits the recapture of the running program's state, which means the stack, heap and data areas are copied to a newly instantiated process of the new version. The new process starts with the existing state. Like PODUS, the system allows the user to code interprocedures that can modify procedures and interfaces, and the programmer can use a state transformation function to

modify the global state. Gupta's most important contribution is a formal framework for understanding whether a dynamic update is valid.

In his dissertation, Hicks identifies two main problems associated with state transfer [OUS 9]. First, as with application-specific state transfer, it is generally not possible to capture OS-level data structures. This includes file descriptors for open socket connections. Secondly, state is address-space dependent, which makes it unusable for a different address space. The limitations of state transfer preclude its application to larger, network-oriented systems such as e-Commerce servers, because connection data (i.e. the socket file descriptor table) is stored in the OS. If losing connections at update time is unacceptable, these limitations rule out a sizeable class of applications [OUS 9].

2.1.12 Popcorn

In his doctoral dissertation, Hicks also develops a dynamic modification technique (Dynamic software updating [OUS 9]) and implements this concept in a language called *Popcorn*. Hicks' method, which is oriented toward dynamic modification in the active networks domain, concentrates on programming dynamically modifiable open-ended systems, particularly when the owner of a software component that will replace an existing component differs from its vendor. The security issues, which can occur when performing dynamic modification, are a main concern of Hicks' research and his methodology focuses on them [OUS 9].

Hick's method imposes restrictions on the software architecture. Here, during the initial design, programmers must develop strategic safe spots that can ensure validity for all the possible future code changes and the associated dynamic modifications. This requirement complicates the design phase. Chapter 5, Section 5.1 provides a description of this issue.

2.1.13 Related Hardware Issues

The concept of a shared library highlights some related hardware issues. For the successful implementation, it is necessary to individualize a shared library to each system. However, when this concept is conceptually applied to real time critical applications, a problem with *cache coherence* [HW 2] results, because a processor will try to keep a duplicate copy of code (that is already present in the memory) used most often in the cache memory. However, if the shared library text is in both the main and cache memories, the question of what happens to this text in the cache memory arises when the library code in main memory is dynamically modified. Although clearing the cache can appear to be a good option, doing so will slow down the system until it caches enough instructions into the memory. Instead, consistency must exist between both copies.

Solving the problem of cache coherency is vital. Keppel believes that such problems exist because there are no provisions in the hardware for handling dynamic modification. The responsibility, therefore, lies with software designers [HW 2]. This can prove complicated, because the programmer must then both modify code and manage

coherency. As hardware has become more sophisticated, however, it has also become capable of handling cache coherency, as evidenced by the use of the MESI protocol in the Intel Pentium processor [HW 1].

2.1.14 Related Operating System Issues

Except in languages like Java, a compiler translates the high-level source code of a program into a corresponding file with machine executable instructions. Whenever this executable file is loaded into memory, then that instantiation of that program is called a process. The execution of a process is generally in accordance with the nature of the operating system. The programmer must take care of two issues whenever he or she attempts dynamic modification.

First, the size of the process' address space can change in accordance with the desired change in the functionality of that process. This can contribute to the need for extra memory during execution. For example, if three new functions are added to an executing process, they will need extra storage space for the additional executable code. At this juncture, if the allocated memory is insufficient, the executing process must request more memory from the operating system. The operating system must, in turn, provide a facility by which an executing process can request such memory.

Secondly, the memory page protection issues need attention. Every executing process includes four different sections: text area, data area, heap area, and stack area. The text area contains executable instructions, while the other three hold the data (See Chapter 3, Section 3.5.1). The memory pages that hold the text area are supposed to be write-protected, but that completely depends on the operating system and the way in which the loader loads the program. For example, operating systems like Unix, Minux, Linux, DOS, and Windows allow processes to modify themselves, with page protection left to the loader or user. These operating systems, with the exception of DOS, do not permit any process to modify the text area of another process without proper authorization, but they do allow a process to change its own text area, if that area has been initially left unprotected. If the text area of the process is write-protected, even the corresponding process cannot modify it. The operating system must provide a mechanism to change the text area irrespective of it being write-protected. *Advanced C: Tips and Techniques* provides a more elaborate explanation of the above example [EX 21].

2.1.15 Process Halting and Process Suspension

This section compares and contrasts process *halting* and *suspension*, both of which are significant to the concept of dynamic modification.

Halting an executing process terminates it forever, with the assumption that to re-execute it, the user must start from scratch with new data and input. If the user intends to resume the process, he or she has to make the loader place the program into the memory and fully restart the execution. Conceptually, halting terminates the executing functionality.

In contrast, process suspension is a temporary maneuver that permits resumption at the point of stoppage, because the process remains in the address space. Operating systems use this method to perform process scheduling. Typically, the process scheduler suspends all other processes to let a single one execute for a predetermined period. After the time expires, the executing process is suspended and the execution of another predetermined suspended process is resumed.

The difference between these two methods is that process suspension and resumption is *history sensitive*, because the address space is retained, whereas process halting and restarting is not. The technique described in this thesis uses the process suspension mechanism to perform dynamic modification.

2.1.16 Runtime Address Space Analysis

Figure 2.1 shows the four memory regions an executing process contains.

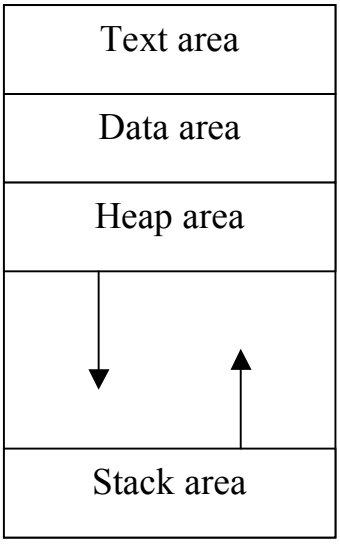


Figure 2.1: Address Space of a Process

All the executable instructions are located in the first region, the *text area*, which is generally readable and executable, but not writable. Any attempt to modify this write-protected area will ensure a memory fault (OS fault). Therefore, if writing is desired, the design process must include a way to modify the pages. The designer (or programmer) must consider this implementation issue, in particular, before designing any dynamic modification technique.

The *data area*, which is readable and writable, stores the global data, constants, and static data. This area is further divided into two parts, consisting of *initialized* and *uninitialized* data. If a static or a global variable is initialized, the associated value is placed in the initialized area or else provisions are made such that a future value associated with that identifier can be placed in the uninitialized area. Constants are placed in the initialized area as well.

The *heap area* provides storage for all values associated with identifiers that have their memory allocated dynamically. Finally, the *stack area* contains activation records, which hold local data of various functions within the process.

During compilation, the compiler fixes the sizes for both the text and data areas, which usually do not vary during runtime. The sizes of both the heap and stack areas, however, tend to change. Generally, all four areas are stored in memory pages.

2.2 Drawbacks of Existing Systems

The central problem with all these systems is that they lack flexibility, because they restrict dynamic modification to certain conditions. For example, dynamic link libraries and dynamic classes (to an extent) restrict dynamic modification to the leaf nodes of the *function call graph* and can only modify at a very coarse grain level.

The user can perform dynamic modification in DAS and DMERT only with the help of some of the operating system features. The state transfer mechanism restricts Gupta's method. Techniques used in Durra and Conic are system specific, because to operate they demand special types of runtime environments.

To date, DYMOS, PODUS and Popcorn seem to be the most flexible methods available for an imperative language, but all of these still restrict the dynamic modification to functions. In addition, as a system, DYMOS is completely dependent upon geographical location, because the integrated development environment (IDE) and the executing process are tightly coupled with each other. Geographical independency is necessary for dynamic modification.

As Gupta suggests, ensuring validity during the dynamic modification process is important [OUS 6, 7]. DYMOS, PODUS, and Popcorn have methods of controlling the dynamic modification process in order to ensure its validity. For every module transition in PODUS, the programmer must perform dynamic modification manually. In DYMOS, the most expressive method to date, the programmer uses a special instructive mechanism containing sequence and selection to tell the system how dynamic modification has to take place. Here, the programmer controls the entire system and communicates the time at which code can be modified and when the type of an identifier can be altered. The use of the programmer-system communication method can ensure the validity of dynamic modification. If it is necessary to perform the dynamic modification at a location far from the programmer, this author argues that even the DYMOS expressiveness is not sufficient.

So far, DYMOS is the only method that provides a formal programmer-system communication path. However, its mechanism can support communications only for code modifications, because DYMOS lacks a formal mechanism that can be employed for type modifications.

Portability is another issue. Under the programmer's control, PODUS and Popcorn can perform remote dynamic modification. PODUS seems to work effectively when modifying programs over a small network. However, if a version of Netscape or the Solaris operating system needs to be modified over the Internet, two problems arise. First, it is highly unlikely that all users will log on to the Internet at the same time. Secondly, users might want to upgrade software at their convenience rather than at the vendor's convenience. In this author's experience, most Chief Information Officers (CIOs) do not want to upgrade their mission critical software unless they are certain the new version will be more economical.

This author, therefore, argues that there should be a mechanism that allows the vendor, working with the client, to configure the dynamic modification process based on the client's business model.

2.3 Dynamic Modification Procedure

Since the executable code and the state of a running process are always compatible with each other, ill-timed dynamic modifications can introduce new code (or functionality) that can disrupt this compatibility. Once performed, a dynamic modification is said to be valid only if the executable code and the state are compatible with each other. The impact of a dynamic modification depends on the associated functionality change. Therefore, whenever a programmer changes the source code of a program for an update, the underlying functionality change decides the complexity of the associated dynamic modification. For example, a simple modification involving a change in the operator (See Figure 1.1) entails changing the associated operand within the address space of the process. However, a type modification is certainly more complex than changing an operand of an executing process. Hence, identifying the impact of a dynamic modification is important. For the purpose of this discussion, *dynamic modification functionality* is the mechanism that dynamically updates the target process.

A code change (update), when dynamically implemented, creates an incompatibility when performed at an inappropriate instant of time. For example, if a tax calculation algorithm is suddenly replaced with another between tax computations, the end result can be incompatible with the input. Therefore, the dynamic modification functionality must modify the calculation algorithm only when the tax computation is not taking place. Chapters 4 and 5 provide a description of the problems arising from *ill-timed modifications*.

After the programmer identifies the ill-timed modifications, it is mandatory to identify when it is safe to perform (*validity condition*) the associated dynamic modification. The programmer must perform dynamic modification when the associated validity condition is met, because this ensures the dynamic modification is safe.

In general, dynamic modification is a procedure that constitutes the following four sequential steps (See Figure 2.1):

1. Identification of modifications
2. Detection of potential incompatibilities
3. Development of a safety mechanism (mechanism that ensures validity)
4. Performance of dynamic modification based on the safety mechanism

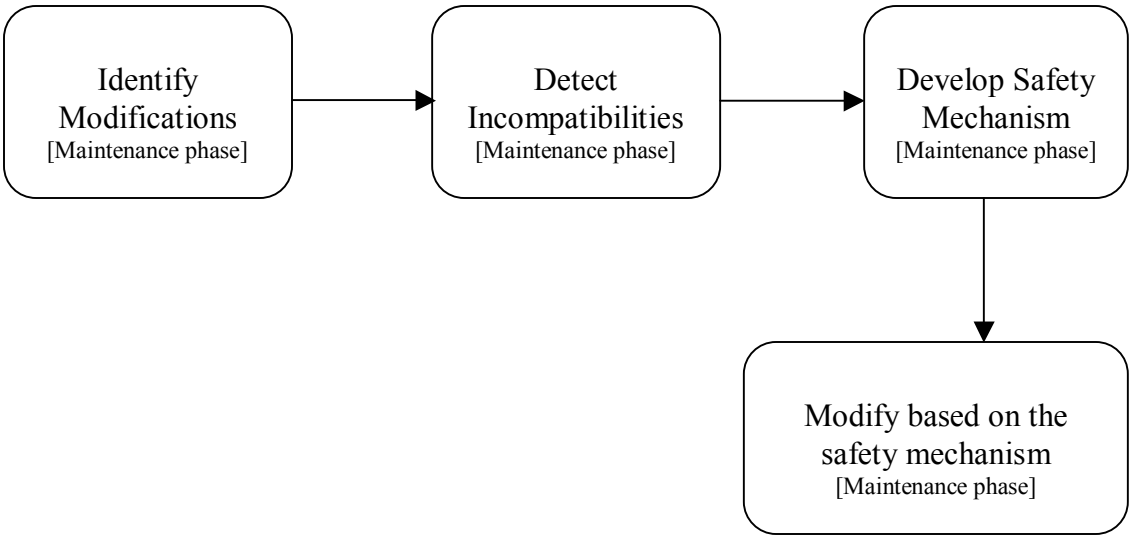


Figure 2.2: Dynamic Modification Procedure

This research investigates the dynamic modification procedure described above. Chapter 3 provides an explanation of how to identify modifications. Chapter 4 concentrates on describing the nature of the incompatibilities that can potentially arise during dynamic modifications. Chapter 5 gives the details of a mechanism for developing the safety mechanism. Chapter 6 reports on the technicalities involved in modifying parts of the address space of an executing process. The next six chapters concentrate on describing this author's technique for dynamic modification and arguing for its validity and usefulness. Then, the final chapter reports on the contributions the research in this thesis makes and the identification of future research projects.

Chapter 3: Granular Functionality

3. Introduction

Before beginning the examination of the first step of dynamic modification (identifying modifications), it is important to study how a program's functionality is organized. In this context, the term *functionality* refers to the characteristics the programmer intentionally assigns to each program module during the design process. To some degree, the programmer assigns the total intended functionality based on his or her own perception of the solution. Irrespective of the programmer's design, a typical program is implemented with the constructs offered by the relevant programming language, the *language constructs*.

Language constructs are the mechanisms programmers use to encapsulate the functionality of their algorithms. These constructs depend on the nature of the language and the nature of the algorithms that the programmer can incorporate into it. It is also possible to infer that language constructs depend on the underlying programming language paradigm. The programmer encapsulates the functionality of the program element within these language constructs as they are syntactically defined. Since an *imperative paradigm* is the main focus of this research, Section 3.1 concentrates on examining language constructs typical of an imperative language. Section 3.2 provides a proposal for a construct hierarchy based on these syntactic properties. Section 3.3 reports on the runtime environment of an executing process with respect to grains. Finally, Section 3.4 concentrates on describing the possible code modifications for an imperative programming language like C.

3.1 Language Constructs

This research uses the term *grain* to represent a language construct. Every language has its own set of grains, which can include control structures, functions, and procedures, among other items. The syntax of the underlying language defines the size of each grain abstractly. If all the grains of a particular programming language are ordered based on their syntactic sizes, the resulting hierarchy is called a *granular hierarchy*. This section explains the different grains in terms of the language constructs used in this research.

Machine language instructions
Machine language instructions constitute the actual machine executable instructions. While every machine language instruction has one purpose or objective, its functionality is atomic. Moreover, such instructions are hardware dependent.

Assembly language instructions
During compilation, the code generator produces assembly language instructions, and an assembler then generates corresponding machine instructions, generally using a one-to-one correlation. An assembly macro, however, produces many machine instructions.

Therefore, the relationship between the assembly language instruction set and the machine instructions set is either one-to-one or one-to-many.

Statements
A statement is a set of *lexemes* arranged in a grammatically defined order with encapsulated functionality. The lexical and the syntactic phases of the compilation process break a statement into a number of lexemes. This research considers a statement to be a lexeme that does not contain additional statements, because a person must abstractly differentiate between the amount of functionality contained in a single statement and a statement that contains further statements. Statements are atomic, and *blocks* are compound statements that contain statement*s*.

Blocks
Blocks encapsulate the functionality of numerous statements. A compound statement list can also constitute a block. "**If-then-else**" and "**while**" loops can be considered blocks. Each block can also contain additional blocks within it.

Functions
A function constitutes a collection of statements and blocks. The main characteristic of a function is that it encapsulates the lifetimes of all identifiers declared within its scope.

Program Unit
A program unit is a collection of multiple functions. The composition of all the functionalities in that unit constitutes the total functionality of the program unit. A program unit can also be considered a subprogram.

Program
A program is composed of units. The union of the functionalities of the program units constitutes the functionality of the program.

3.2 Granular Hierarchy

In the granular hierarchy, machine instructions are the smallest grains. An assembly language instruction translates into one or more machine instructions. A set of assembly language instructions corresponds to a *statement*. This explicitly means that a statement can be decomposed into machine instructions, as can a block, since it consists of a combination of statements. Similarly, it is possible to demonstrate that all the other grains can be thus decomposed. In essence, machine instructions, which are the basic building blocks of a program, are also known as *atoms,* because they are the smallest possible indivisible grains. Logically, then, it can be stated that the size of a grain depends on the general number of atoms into which it translates.

If the number of constituent atoms of each grain in decreasing order is considered from an abstract programming language perspective, it is possible to obtain a *grain hierarchy*, as shown in Figure 3.1.

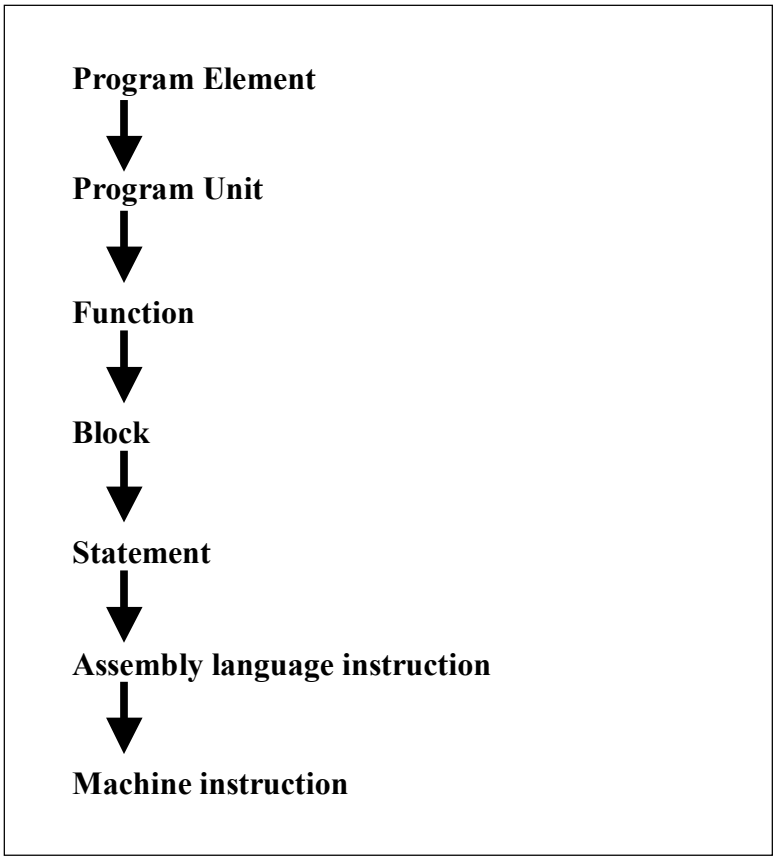


Figure 3.1: Decreasing Order of Granular Size

Every grain in the hierarchy has a one-to-one or one-to-many relation within the immediate smaller grain. Figure 3.1 shows a descending hierarchy in which grain size decreases to machine instruction, the smallest possible unit. The largest grain, with the maximum number of atoms, is the program element. Next in this descending hierarchy comes the program unit, which can be decomposed into functions, functions into blocks, blocks into statements, and so forth.

3.3 Runtime Grain Analysis

Runtime grain analysis, a simple test that can at any time categorize the grains present within the address space of an executing process, demonstrates that it is possible to categorize the grains by suspending the program and examining its address space. Whenever the programmer performs this analysis, it is possible to observe two kinds of grains, *active* and *inactive* (see Figure 3.2).

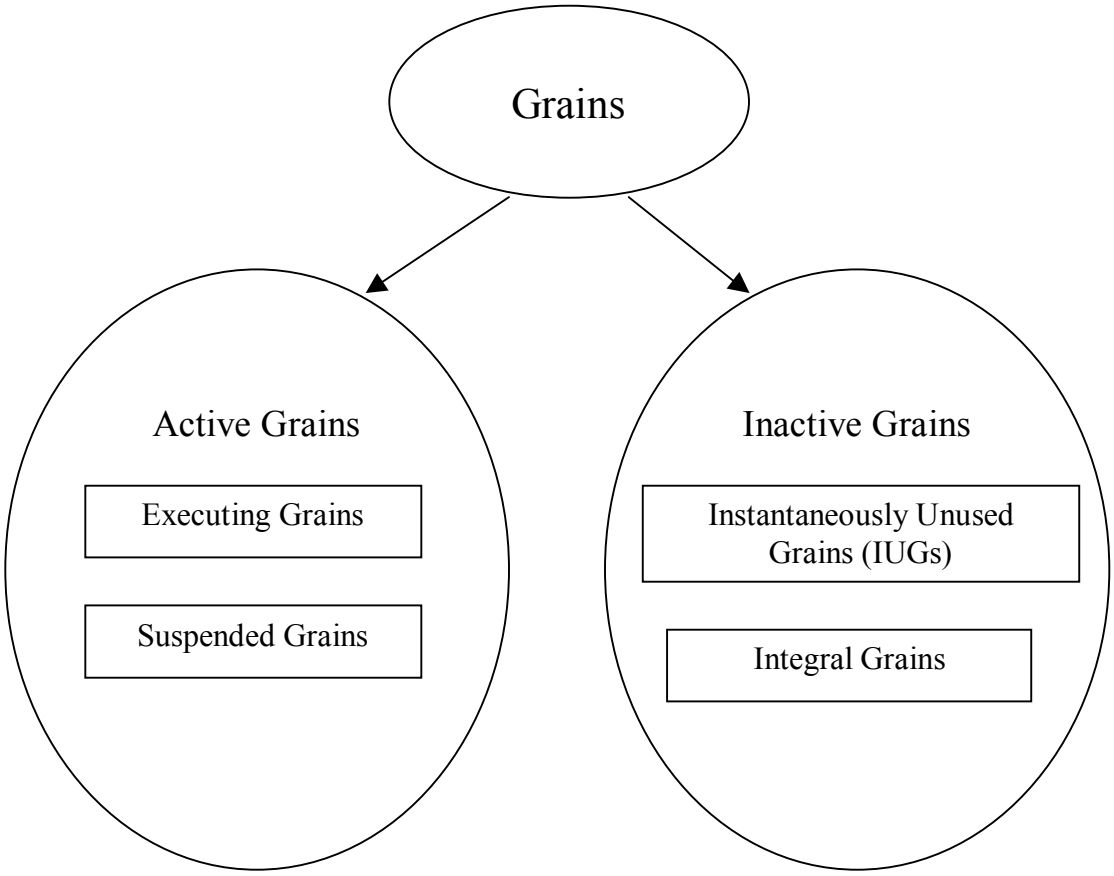


Figure 3.2: Grain Categorization

For the purpose of this discussion, the *execution pointer* is defined as an imaginary pointer that moves around the control flow graph of a program. The execution pointer always points to the memory address that is associated with the instruction counter. Thus, if this memory address changes, the execution pointer moves and points to the associated new memory address. On the one hand, if the execution pointer is outside the scope of a grain, then that grain is an inactive grain unless it does not have a stack frame that is part of the instantaneous call sequence in the stack. Active grains, on the other hand, have an associated state at the instant the programmer performs the runtime grain analysis. There are two kinds of active grains: *executing* and *suspended*.

An executing grain is the grain whose very execution is suspended during the instant when the programmer performs runtime grain analysis. In other words, during analysis the states of these grains are stored in the stack frame that is at the top of the stack. At any point of time, the execution pointer can be within the scope of one machine instruction. This machine instruction is encapsulated within the scope of a conceptual assembly language instruction, which in turn is encapsulated within a statement.

As a result, it can be implicitly stated that there can be only one executing program unit, one executing function, one executing block, one executing statement, and so forth. At the instant at which the programmer performs the runtime analysis, the execution pointer

resides within the scope of a set of grains of unique sizes, the *executing grain set* (EG). The set EG contains seven grains of different granular sizes or, in other words, a grain from each level of the granular hierarchy. It is possible for this argument to be different for concurrent threads, but it holds for this research, because this research focuses on single threaded imperative programs.

Every grain can be modified only if the address associated with the instruction counter is outside the domain of a special region called the *integral section*. If a grain is modified when the execution pointer is within the scope of the corresponding integral section, incompatibility can arise. Chapter 5 provides a discussion of this issue.

There are two categories of inactive grains, *integral* and *instantaneously unused*. On the one hand, if the execution pointer is within the domain of the integral section of a grain, then the grain is called an *integral grain*. On the other hand, if the executing pointer is outside the domain of the integral section of a grain, then the grain is an instantaneously unused grain (IUG). In general, the *dynamic modification functionality* must modify IUGs at any instant, because they do not have an associated state, but it must delay modification of integral grains until the instant when the execution pointer comes outside the scope of the associated integral section. Chapter 5 reports further on the concept of integral section.

3.4 Kinds of Modifications

This section provides a description of the various possible modifications that the dynamic modification functionality can perform on the source code. This corresponds to the first step within the dynamic modification procedure (See Section 2.3). Before examining the nature of these changes, it is important to study the conceptual organization of the way in which code and data are related to each other. This section briefly describes the relationship between data and code before introducing the two main kinds of modifications.

Data associated with a process is stored within specially instantiated data objects, instances of a particular type. The type of the data object defines the way in which the corresponding data object instantiation stores information. For example, the way it is stored in an object of type integer is different from that of an object type float.

Every data object has a lifetime. If this lifetime is equal to that of the program, then the object is stored within the data area; if it is equal to that of the encapsulating function, then it is stored within the stack. The programmer can store data objects within the heap area by explicitly allocating memory during the runtime (`malloc()` in C language).

During development, the programmer associates identifiers, either explicitly or implicitly, with data objects and then implements functionality within the program by manipulating identifiers. During compilation, whenever the compiler encounters such an identifier, it generates a corresponding type code. In informal terms, type code is code that can access the information stored within the data object of the respective identifier. The programmer

can use a single identifier at many different places within the source code and, with each instance, the generator creates the equivalent type code. In this way, data represented within the corresponding object becomes accessible.

In this scenario, there are two kinds of modifications: code changes and type changes. The first kind is the code change. The programmer can add new code and modify or delete the existing code. A code change can be informally defined by using granular hierarchy (See Figure 3.1). A code change involves modification of the target code alone, although relocation can be required.

The second kind of modification, type change, is more complicated than the first. Whenever the type associated with the data object needs modification, all instances of the type code of the corresponding data object have to be modified. For some type modifications, the programmer has to make explicit source code modifications and for the rest, the compiler generates object code that is different. For example, if a global variable of type integer is changed to a float, the compiler (without the programmer's knowledge) generates an entire new set of type codes that access the associated data object. The programmer must consider this important issue whenever he or she wants to dynamically modify the type associated with a variable. In this scenario, data objects can be divided into two classes of objects based on their lifetimes and the area within which they are stored: they are either *stack variables* or *data* and *heap variables*. Modification of data objects in general involves adding new identifiers, deleting existing ones, or modifying their types.

Type changes also include changes to the interfaces of functions (or modules). In general, functions are associated with input and output types; in fact, a function itself is a type. A function can take in arguments and return a value. The programmer develops a function in such a way that it assumes control whenever the execution pointer enters the scope of associated function calls. Now, modification of the interface (input and output) type causes the corresponding modification to become complex. It is important to study the concept of incompatibility before examining the solutions for type changes, which Chapter 6 concentrates on describing (See Section 6.2).

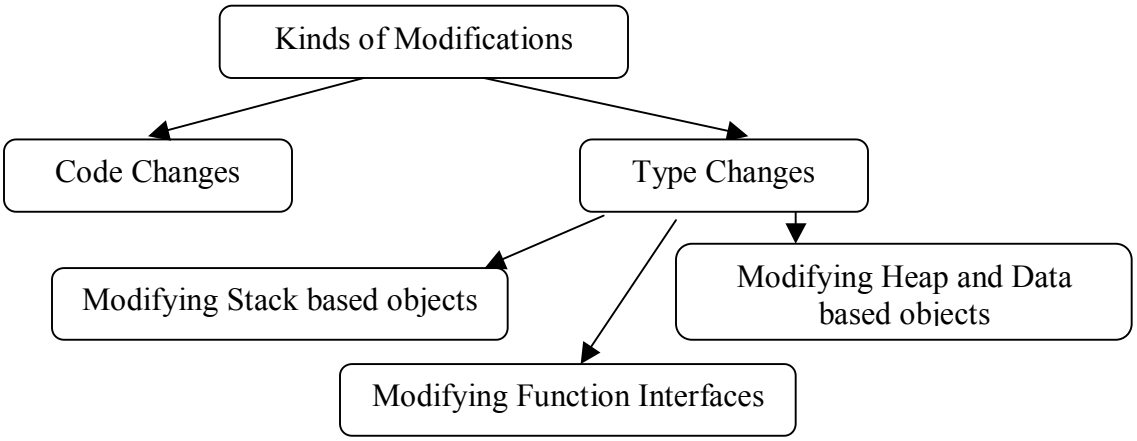


Figure 3.3: Kinds of Modifications

Chapter 4: State Compatibility

4. Introduction

Dynamic modification entails altering the functionality of an executing process without halting it. Whenever the programmer performs modifications to the source code of the associated executing process, he or she initiates the dynamic modification functionality and instructs it how to modify software. Based on the programmer's instructions, this dynamic modification functionality modifies the executing process. Its success rests in part with the concept that a process's executable code is compatible with its state. This research examines the state of an executing process from the perspective of such dynamic modification.

Whenever an attempt is made to modify executable code, there is always the possibility that the new or modified code can become incompatible. Hence, when dynamic modification is performed on the functionality of an executing process, care must be taken that the new functionality remains compatible. Gupta, therefore, describes the term *validity* [OUS 7] as:

> *An on-line change in the process P from configuration Π to Π' at time t (in state s) and using the state mapping S_m is valid if after the change, P is guaranteed to reach a reachable state of Π' in a finite amount of time* [44].

Hence, after dynamic modification, the new code and the state are compatible if, and only if, the process reaches a desired state after an arbitrary amount of time. This chapter gives a detailed discussion of the challenge this situation presents to researchers in the field of dynamic modification.

The state of an executing code is defined as set *S*. It contains the related data values dynamically generated as a result of the execution of code. Within the scope of the program, state is always associated with the location of the execution pointer, as well as with the input. The term *compatibility* thus has multiple focii; as Gupta explains [OUS 7]:

> *Compatibility between the executable code and its state exists if and only if the executable code is associated with a valid set of values stored in the state S such that the execution of that code makes the executing process behave in the way the programmer prescribes it to behave* [17].

The state can be further divided into two categories: *internal* and *external*. The underlying operating system, as described below, draws the distinction between both.

An internal state consists of the data that at least one machine instruction of the executable code can directly access. The code does not depend on any of the operating system features in order to access this data. Since this research emphasizes a POSIX

operating system, the internal state can be hypothetically stated as the *data present within the address space of the process*. This internal state generally consists not only of the data stored in the stack, heap, and data areas, but also of *hidden data* stored in the stack pointer, base pointer, instruction counter, and registers of the corresponding executing process.

If this internal state has to be copied to the output, the executing process must use system calls. Text display in Linux exemplifies this. Typically, the executing process uses system calls (e.g., `printf`) meant for output and passes the text data that has to be displayed as an argument. These OS calls ensure that the data is copied to the output. System calls store the values of the arguments at some arbitrary location and perform predetermined computations. Irrespective of the performed action, after the corresponding output, the executing code of the process cannot directly access this exported data. Another situation specific input system call (e.g., `scanf`) must be used to retrieve data from the outside into the address space. This is the nature of a realistic multi-tasking operating system. As there is an assurance that the operating system controls the usage of system resources, a POSIX-compatible OS also falls within this category.

The external state of a process is the set of all data stored outside the virtual memory allocated to the address space of the associated process. Operating system calls can access this data, but situation-specific system calls, if they exist, are the only means of modifying this data. This research considers process-specific data to be the internal state of the operating system.

Once dynamic modification is complete, the new executable code must be compatible with its state. However, it is not mandatory for an executing process to have an external state. Assume that the term *visible dat*a refers to data contained in the stack, heap, and data areas. Each visible datum is associated with an identifier that has defined properties, which depend on the language used to develop the corresponding program. The relation between the domain of identifiers and the domain of visible data is generally one-to-many, one-to-one or many-to-one. As the compilation environment generates this mapping, it is essential that information regarding data values and machine instructions is available.

The same argument cannot hold for the external state, because neither the executing process nor the compiler has information about the external state. Instead, the operating system stores this external data in some arbitrary, predefined memory space. Moreover, the external data takes on meaning when the recipient views it. For example, an output displayed on the screen conveys some meaning to the user, and the user is the only one who can decide whether or not that output is valid. There is no way for either the process or the operating system to understand this meaning.

The research in this thesis makes the argument that it is possible to develop an algorithm that can avoid potential incompatibilities between the executing code and its state, if, and only if, it is possible for the executable code to access the state and the conditions for

compatibility. If either of these two conditions fails, ensuring compatibility by computational means becomes complex.

To reiterate, a dynamic modification is said to be valid if, and only if, the executing process behaves as expected afterward. Validity can only exist if the executable code is compatible with the state. Sections 4.1-4.3 provide descriptions of the three possible scenarios that tend to disturb this compatibility during a dynamic modification.

4.1 Type Incompatibility

In a programming language, every identifier is associated with a type. In any language, an identifier can interact with another identifier if, and only if, the semantics of that language permit it. For example, after examining the statement "`a := b`," it is possible to infer that the variable "`a`" interacts with "`b`." A strongly typed language permits only identifiers of the same type to interact with each other, while a weakly typed language permits identifiers of different types to interact with each other under certain well-defined scenarios. This interaction is based on the semantic properties of the language being used. For the purpose of this discussion, the term *interactive set* is used to describe a set that contains all the identifiers that interact with the corresponding ones under observation. Every identifier has its own interactive set, and each identifier is an element of its interactive set. In general, analyzing the lifetimes of the identifiers within the source code can statically detect this interaction.

Once the program starts, every identifier interacts with the elements of its interactive set using a protocol mechanism that governs interaction according to the semantic properties of the underlying language. Type incompatibility during a dynamic modification occurs whenever there is a protocol or type mismatch between an identifier and at least one member of its interactive set, irrespective of the nature of the underlying language.

For example, assume that version 1 of a program has an `integer` identifier "`a`" that has an interactive set containing: "`b`," "`c`," and "`d`". Assume that "`b`," "`c`" and "`d`" are also of the type `integer`. The identifier "`a`" can interact with its interactive set using operators such as `:=` ,`>=`, `<=`, `=` etc. This interactive set can be represented as

{ (int)a, (int)b, (int)c, (int)d }

Assume that the programmer modifies the type of "`a`" and its interactive set to float in a subsequent version. The corresponding new interactive set can be represented as

{ (float)a, (float)b, (float)c, (float)d }

Now, the executing process of version 1 contains an integer interactive set, and the new version, the patch, consists of a float interactive set.

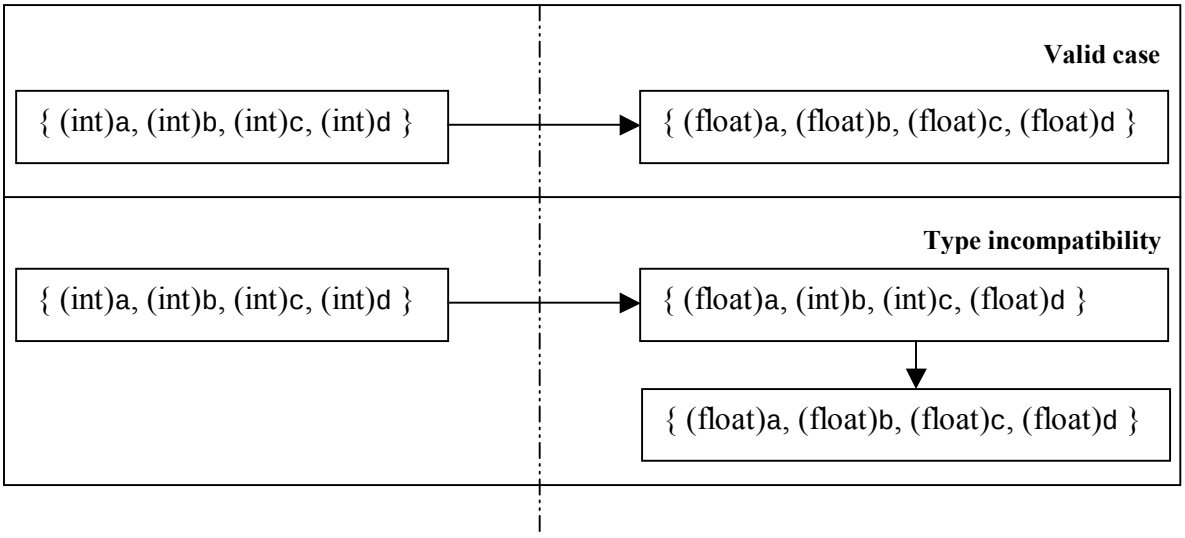
Whenever the dynamic modification functionality performs dynamic updates at an inappropriate time, incompatibility occurs. During the process, there is every possibility

that the interactive set present in the executing process simultaneously contains unique identifiers of different versions. This implies it is possible to have an interactive set such as

{ (float)a, (int)b, (int)c, (float)d }

As soon as this situation occurs, whenever "a" interacts with "b" or "c," a type mismatch results. When the dynamic modification functionality updates at inappropriate times, it generates an intermediate interactive set that, as time passes, converts into a proper interactive set (See below).

{ (float)a, (float)b, (float)c, (float)d }



Before the intermediate interactive set completely converts into a valid interactive set, certain improper type interactions lead to a type mismatch. Although it is imperative to avoid this situation, the only feasible method for doing so is to modify all the identifiers of an interactive set at the same time. These interactive sets can be generated during the compilation time.

Figure 4.1 shows a transition that can potentially create type incompatibility. The assembly language instructions in the Linux operating system on an x86 architecture that correspond to statement 5 (line 5 in Figure 4.1) are shown:

```
Mov EAX, [:::] // relative stack location corresponds to the variable "b"
Push EAX
Call  tempfun
Pop EAX
Mov [:::], EAX // move the value to the stack location that corresponds to "c"
```

Assume that just before the call instruction of the assembly language equivalent to the 5th statement, the execution pointer is within the scope of line number 5 (Figure 4.1). Assume also that the `push EAX` instruction has just been executed and that the execution pointer is about to enter the scope of the next instruction when the dynamic modification functionality attempts dynamic modification. During the modification phase, the dynamic modification functionality identifies the execution pointer to be within the scope of the 5th statement.

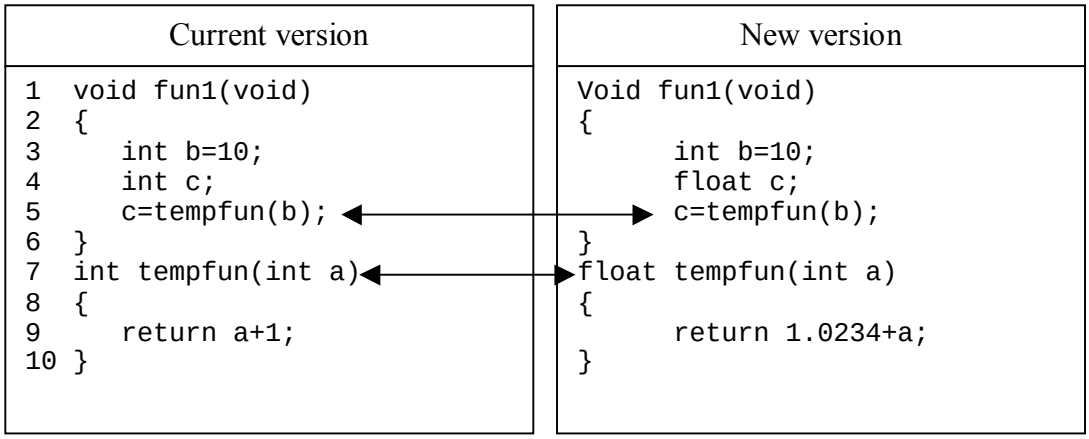


Figure 4.1: Type Incompatibility

This situation now makes statement 5 an executing grain. It is important to note that the executing pointer is about to enter the scope of the '`tempfun()`' function exactly when dynamic modification is attempted. Therefore, the dynamic modification functionality cannot modify statement 5, because some of the instructions of the 5th statement have already been executed and it is irrational to roll back the execution. Of course, execution roll back can be done with this example, but roll back cannot be generalized for all cases. The dynamic modification functionality can only modify statement 5 when the execution pointer enters the scope of statement 6.

Since the function '`tempfun()`' is an inactive grain, and since it does not lead to a binary mismatch, the underlying software system can modify it immediately. A binary mismatch can occur if the executable code of a function cannot properly access the declared data within the visible data. In general, there is a higher possibility for a binary mismatch to occur when an executing grain is modified. Therefore, the dynamic modification functionality delays the modification of the fifth statement and instead modifies the '`tempfun()`' function. The application of dynamic modification also causes the interface of '`tempfun()`' to alter. The new '`tempfun()`' function reads an integer and returns an object of the type float.

As soon as the execution resumes, the new version of '`tempfun()`' returns a floating point representation. At this stage, statement 5, which has remained unmodified and can handle integers, cannot handle the floating-point value, and a type mismatch occurs. The underlying software system needs to avoid this scenario. A simple remedy would be to modify statement 5 and '`tempfun()`' simultaneously, as an atomic process. It is also

possible to add a modification of statement 4 to this atomic modification process to ensure validity.

4.2 Internal Semantic Incompatibility

During execution, almost every grain brings in a change to the state of the process. From an abstract perspective, the functionality of the executing grains has a direct effect on the state; in fact, most of the grains have an effect on the state data. An examination of this situation reveals that a grain brings some change to a datum and that this change can depend on the initial value of the datum as well. For example, the grain "`temp++`" increases the value of "`temp`" by 1. The final change the grain brings in is dependent on the functionality of the grain as well as the initial value stored in the identifier "`temp`." In other words, the change can be history-sensitive.

Until its lifetime ends, a datum holds a particular representation generated and altered by all the grains that have access to it. The value can also depend on the other data entities. For example in the statement "`a := b + c,`" the value of "`a`" depends on "`b`" and "`c`." In this case, it is possible to draw a data flow graph showing that the data flows from "`b`" and "`c`" to "`a`." It is also possible to extend this data flow graph to encompass all identifiers that contribute to the values that flow within it. All of the grains that can access at least one identifier within this data flow graph together constitute a set, termed a *semantic set*. In fact, the value propagated within this data flow graph results from the interdependent behaviors of the semantic set's grains, also known as *semantically compatible grains*. Such grains interact with each other using the data that passes within the graph.

During dynamic modification, if modified grains of different versions of a semantic set affect the data, it is possible for an invalid situation to arise. For example, if it is necessary to modify a version of the semantic grain set {`A`,`B`,`C`,`D`} to a different version, {`A`` `,`B`` `,`C`` `,`D`` `}, then all the grains of the set are to be modified. During dynamic modification, if there is a situation where the semantic set comprises grains of different versions, as, for example, in {`A`,`B`` `,`C`,`D`` `}, potential incompatibility arises, mainly because of data corruption within the flow. Since each grain affects the data flowing within the path in a manner the programmer prescribes, when different versions of grains interact, data become corrupted. The corrupted data create this invalid situation known as *internal semantic incompatibility*. As a result, corrupted data can linger for an unspecified time and eventually cause a system crash.

For example, Figure 4.2 represents the two functions of an initial version, Version 1. During the execution of Version 1, the following steps take place:

1. `fun1()[version 1]` calls `fun2()[version 1]` and passes a value 10.
2. `fun2()[version 1]` computes the argument and returns the value 100.
3. This 100 is stored in the location represented by the identifier `GlobalVar` in `fun1()[version 1].`

4. A value of 10 is added to this `GlobalVar` and a value of 110 is displayed on the screen.

```
void fun1(void)                          int fun2(int dumpVar)
{                                        {
      int GlobalVar;                           return dumpVar*dumpVar;
      int tempVar;                       }
      tempVar = 10;
      GlobalVar = fun2(tempVar);
      GlobalVar = GlobalVar + 10;
      Printf("%d", GlobalVar);
}
```

Figure 4.2: Sample Version 1

Figure 4.3 depicts the programmer's modification of these two functions in a subsequent version, Version 2.

```
 void fun1'(void)                    int fun2'(int dumpVar)
 {                                   {
       int GlobalVar;                      return dumpVar*dumpVar+10;
       int tempVar;
       tempVar = 10;                 }
       GlobalVar = fun2(tempVar);
       Printf("%d", GlobalVar);
 }
```

Figure 4.3: Sample Version 2

In this new version, the net result of `fun1'()[version 2]` and `fun2'()[version 2]` is the same as `fun1()[version 1]` and `fun2'()[version 1]`, but the way the result is computed is different. During the execution of version 2, the following steps occur:

1. `fun1()[version 1]` adds 10 to `GlobalVar` after getting the squared number from `fun2()[version 1]`.
2. Similarly, `fun1'()[version 2]`directly displays the number that is obtained from `fun2'()[version 2]`. Here `fun2'()[version 2]` adds 10 to the squared number before it returns the computed value.
3.

Both `fun1()[version 1]` and `fun2()[version 1]` represent a semantic set. Similarly, `fun1'()[version 2]` and `fun2'()[version 2]` represent a different version of the same semantic set.

Assume then that the smallest grain that can be modified is a function. Assume that the execution pointer is within the scope of `fun1()[version 1]` when the dynamic modification is attempted (see Figure 4.4). Say `fun2()[version 1]` has not yet been called. Here, both `fun1()[version 1]` and `fun2()[version 1]` have to be modified with `fun1'()[version 2]` and `fun2'()[version 2]`, respectively.

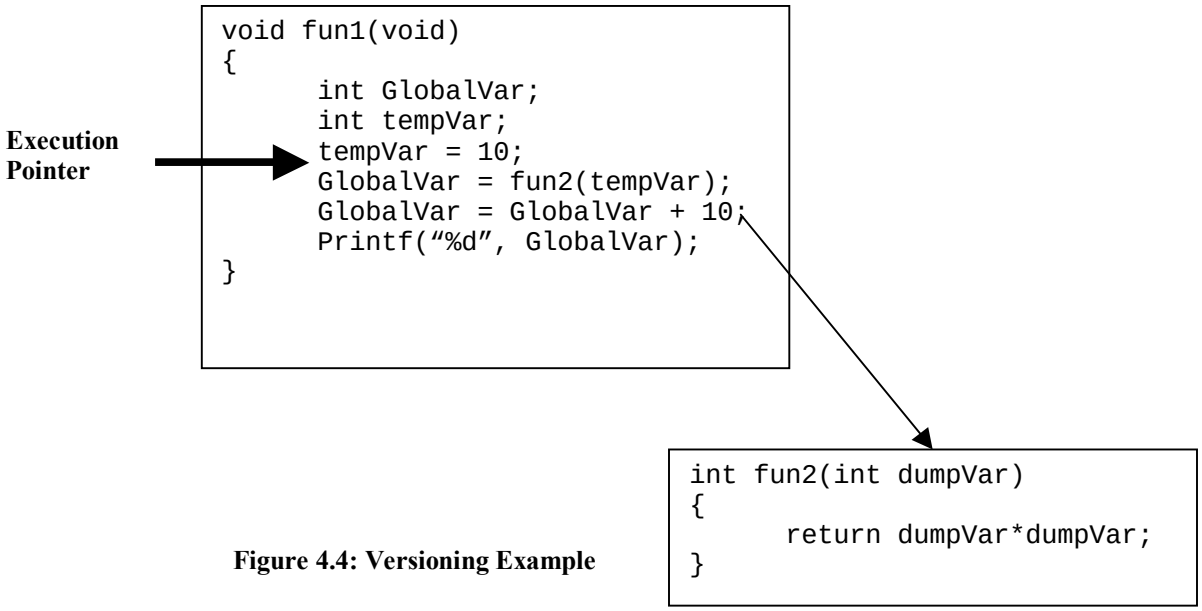


Figure 4.4: Versioning Example

As granularity is restricted to functions, `fun1()[version 1]` cannot be modified because it is an executing grain. `Fun2()[version 1]` can, however, be modified to `fun2'()[version 2]`, as `Fun2()[version 1]` is an inactive grain. The modification of `fun1()[version 1]` has to be delayed until the execution pointer exits its scope. If there is a delay in the modification of `fun1()[version 1]` and if `fun2()[version 1]` is modified to `fun2'()[version 2]`, when the program resumes execution after this modification, then the following steps occur:

1. `fun1()[version 1]` calls `fun'2()[version 2]` and passes a value 10.
2. `fun2'()[version 2]` returns a value of 110.
3. `fun1()[version 1]` adds 10 to this value making it 120.
4. Finally the value 120 is displayed.
5. As soon as the execution pointer comes out of the scope of `fun1()[version 1]`, the delayed dynamic modification of `fun1()[version 1]` to `fun1'()[version 2]` takes place.

Since the computation involved in the value 120 is invalid, the dynamic modification is equally invalid. Moreover, compatibility between the executable code and the state is disturbed.

If the underlying software attempts dynamic modification when the execution pointer is outside the scope of `fun1()[version 1]`, it will not create any inconsistencies. Since most incompatibility problems arise as a result of ill-timed modification, this author argues that it is necessary to address this issue in order to ensure the validity of the entire process.

4.3 External Semantic Incompatibility

External semantic incompatibility is comparable to internal semantic incompatibility, which occurs whenever the code of an executing process becomes incompatible with the internal state. Similarly, *external semantic incompatibility* occurs whenever the code becomes incompatible with the external state.

The external state represents data outside the address space of the corresponding executing process. Typically, the executing process copies data out of its address space to convey some information to the outside environment, the user, the programmer, or the like. This environment intuitively expects a desired external state from an executing process and performs arbitrary actions based on that expectation. External semantic incompatibility arises whenever the environment accesses the external state of a process from which it cannot infer the correct meaning.

External semantic incompatibilities also occur because of improper input/output (I/O) interactions. For example, an executing process interacts with the external environment using input, output, or both. A typical environment-process interaction consists of input and output combined. The external environment interactions, be it users or other software programs that interact with the target application either directly or through a network, make the process, and the results of its computation dependent. If dynamic modification is performed after input, a situation can arise where the external environment realizes that the output does not contain valid information.

The difference between external and internal semantic incompatibility is that the former generally does not have any lurking corrupted internal states. Instead, the process misbehaves with the outside environment. Most of the abnormal behavior is generally temporary in nature. Therefore, it is tempting to ignore the seriousness of external semantic incompatibility. However, programmers must not succumb to this temptation, because external semantic incompatibility can create I/O incompatibilities that have serious consequences.

For example, the routine shown in Figure 4.5 calls a function that displays a line at a desired location on the screen. Assume that `Display_line_on_screen(int)` takes a row number and displays a red line on the screen at the vertical location as described by the row number. The main functionality of the example shown in Figure 4.5 is to paint the screen with red.

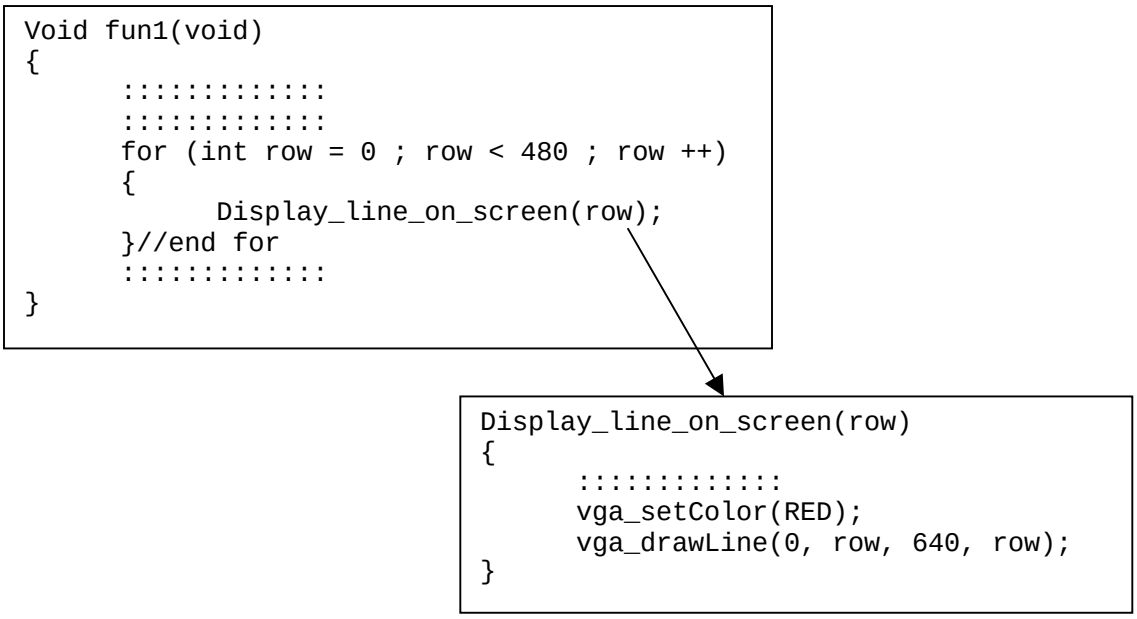


Figure 4.5: Paint Problem

Here, the graphic library call `vga_setColor()` takes in a color value and sets the graphic driver accordingly. This graphic driver draws every pixel with the specified color until `vga_setColor()` resets it with a different color. It is important to note that both the graphic functions `vga_setColor()` and `vga_drawLine()` are graphic library function calls. Now, suppose the dynamic modification is applied during the execution of the iterative `for` loop. Assume that the value associated with the identifier '`row`' is 200. Hence, 280 more iterations must occur before the paint process is complete. At this point, the screen displays 200 rows of red colored lines. Say the execution pointer is about to enter the scope of the `Display_line_on_screen` function. The dynamic modification technique at this instant overwrites the `Display_line_on_screen` function with a new version that displays a blue icon (`vga_setColor(BLUE)`). Once the program resumes execution, the loop performs the rest of the 280 iterations. Finally after the loop completes its execution, the display contains 480 rows, with the first 200 rows filled with red lines and the rest of the 280 rows with blue lines.

Since this example affects the way a program outputs data, while the internal state remains unaffected, this is called an *external semantic inconsistency*. Since this inconsistency depends on the way in which the executing process presents data to the output, which is difficult to detect by computational means, it is virtually impossible to write an algorithm to detect such a problem before it occurs. Essentially, the programmer is the only one who can detect and avoid such inconsistencies.

In summary, it is important and necessary to avoid all the described incompatibilities in order to perform a valid dynamic modification. Chapter 5 provides a description of mechanisms that can detect and solve these incompatibilities.

Chapter 5: Concept of Validity

5. Introduction

As the discussion in Chapter 4 points out, the root cause of incompatibilities is generally a mistiming of grain modification. More specifically, the dynamic modification of a grain can lead to incompatibilities if it occurs when the address contained within the instruction counter points to a location that is inside a region of the address space called the *integral section*. In Figure 5.1, where the dots represent grains and the line connecting the dots the flow control, the gray-shaded rectangular region depicts the integral section of the second statement in the sequence.

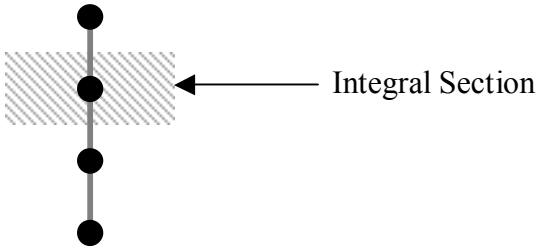


Figure 5.1: Integral Section (Graphical Representation)

Every program has a *control flow graph* (CFG) that depicts the flow of control within it by showing every possible path that can be executed from the beginning of the program until its end. The integral section of a grain is a set of an arbitrary number of disconnected contiguous regions within the control flow graph, within which the execution pointer must not be present when the underlying dynamic modification functionality performs dynamic modification on the original grain. The cardinality of the integral section can be zero or greater. In the first version, grain A calls grain B. As soon as grain B completes execution, the control returns to grain A. If the programmer wants to modify the second version to say grain A calls grain B and grain B calls grain C, the integral section of grain C is a null set when a new grain is being added to the executing process and the call to that grain is present within another grain that is about to be updated to a calling grain. Every grain that needs modification has a corresponding integral section.

In Figure 5.2, which is derived from Figure 4.5, the integral section for the `Display_line_on_screen(row)` function is in the scope of the iterative `for` loop within which the function call is embedded. Hence, the dynamic modification functionality can only modify the `Display_line_on_screen(row)` function when the execution pointer completes the iterative `for` loop. If the dynamic modification functionality carries out the modification when the execution pointer is within the integral section, external semantic incompatibility occurs.

```
void fun1(void)
{
      ::::::::::::
      ::::::::::::
   for (int row = 0 ; row < 480 ; row ++)
   {
         Display_line_on_screen(row);
   } // endfor
      ::::::::::::
}
```

```
Display_line_on_screen(row)
{
      vga_setColor(RED);
      vga_drawLine(0, row, 640, row);
}
```

Figure 5.2: Integral Section for the Paint Problem Shown in Figure 4.5

In another example, the programmer dynamically modifies the interface of a function, where the he or she also has to modify all the corresponding function calls. Hence, the dynamic modification functionality can modify the intended new function only if the execution pointer is not within the scope of the corresponding old function, which is set for replacement, and also not within the scope of the statements (function calls) that call the corresponding function. Here the integral section constitutes the scope of the target function and also the scopes of the grains that call the target function. Figure 5.3 depicts this modification.

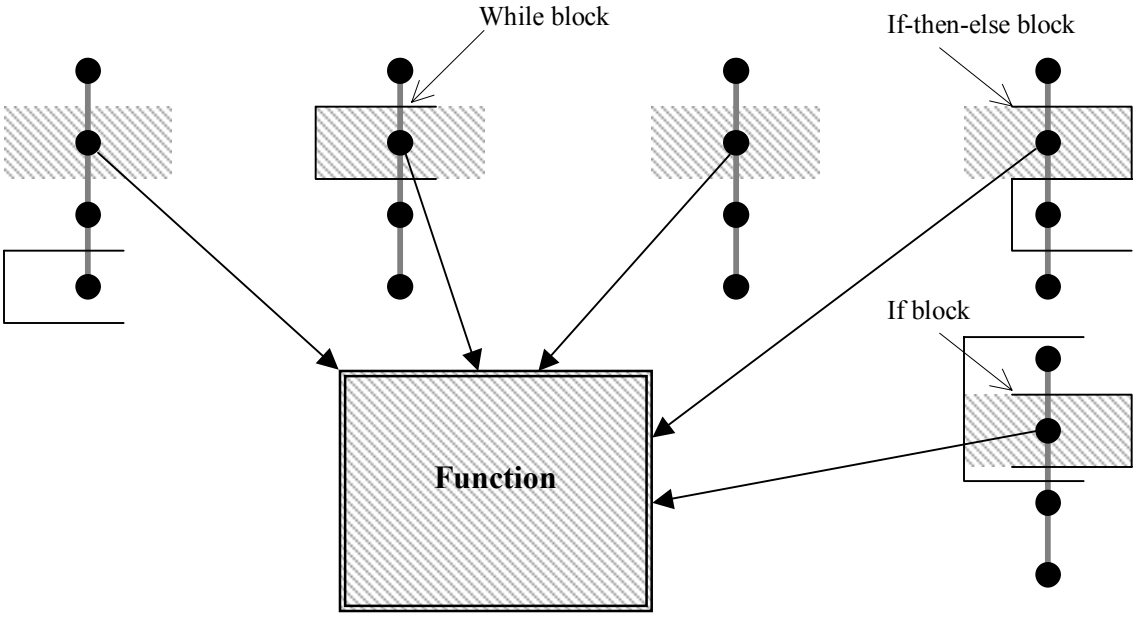


Figure 5.3: Integral Section for the Modification of the Interface of a Function

Accordingly, the research in this chapter examines the underlying issues behind the identification of integral sections and provides informal arguments to justify them. Formalization of these arguments is left as future work

This chapter is divided into three sections. Section 5.1 reports on the two approaches that can be used to solve the timing problem, one of which this research supports. Section 5.2 provides a discussion of the components of an integral section. Finally, Section 5.3 concentrates on explaining the *integral semantic algorithm*, which helps the programmer identify and construct the integral sections.

5.1 Timing Mechanisms

With respect to the grain requiring modification, the programmer can divide the address space of a process into two distinct regions: the *integral section* and the *safe section*, which are always completely disjoint. The safe section constitutes the entire address space, excluding, of course, the integral section, where the dynamic modification functionality can not modify the grain safely. A *safe spot* represents any memory location within the safe section. It is necessary to guarantee safe modification, because only in that situation will it remain valid.

There are two mechanisms that can solve the timing problem. Section 5.1.1 provides a description of the first mechanism, the *explicit safe spot mechanism*. Section 5.1.2 reports on the details of the second method, the *implicit safe spot mechanism*, which this research supports.

5.1.1 Explicit Safe Spot Mechanism

The explicit safe spot mechanism enables a programmer, during program design, to identify a series of strategic spots that can serve as possible safe spots for future dynamic modifications. To accomplish this, the programmer must hard code the strategic spots within the program's source code and strategically locate such spots so that the execution pointer reaches them frequently. This involves designing the software architecture accordingly. Whenever the programmer modifies the program, he designates one of the strategic spots as the safe spot where the corresponding dynamic modification should take place. As soon as the execution pointer reaches the designated safe spot, the dynamic modification functionality can perform the update. Although similar in concept, the explicit safe spot mechanism follows a slightly different dynamic modification procedure from the one described in Chapter 2, Section 2.3 (See Figure 5.4).

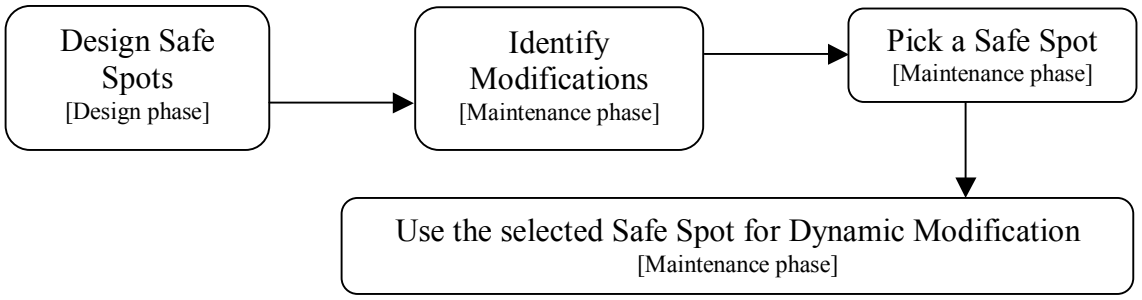


Figure 5.4: Dynamic Modification Procedure for Explicit Safe Spot Mechanism

The dynamic classes technique used in C++ [SL 7] and Java [SL 8] exemplifies this mechanism. During the design phase, the programmer ensures that every class gets instantiated frequently. Whenever an old class needs modification, a special class loader delivers the software handle of the new version to the executing process. With the software handle, the new version of the class will be instantiated whenever the corresponding old one would have been.

The advantage of this method is the simplicity of its dynamic modification functionality, which refers to the software component responsible for updating the executing process. However, the method's disadvantage is its dependency on software architecture. The programmer has to explicitly design the program in a particular way; hence, the technique possesses a learning curve. Moreover, during the design phase, the programmer concentrates on developing the required functionality, which is a complicated procedure in itself. If he or she is also required to design the software architecture accordingly, his or her task becomes doubly difficult.

In the case of imperative languages, the only alternative for the programmer is to design the program so that its functions can be replaced exactly in the same way as the dynamic classes, but this approach becomes more complicated as the size of the program increases. Hicks' technique seems to present an extension of the dynamic classes technique to the imperative paradigm [OUS 8, 9]. Here the programmer needs to specify the exact spot where dynamic modification can occur.

5.1.2 Implicit Safe Spot Mechanism

According to the implicit safe spot mechanism, when modifying his or her source code, the programmer must identify the integral section for every proposed dynamic modification. Section 5.2 provides a description of the components of the integral section and its identification procedure. Once the programmer constructs the integral section and identifies a safe spot that is comfortably outside it, the underlying system must ensure that dynamic modification occurs when the execution pointer reaches that location. It should also ensure that the execution pointer reaches the safe spot often. A safe spot need not constitute just a location, but can also contain a state dependent condition. The representation of the safe spot along with a state dependent condition (if it exists) is called a *timing rule*. A timing rule for a particular functionality change describes the instant of time when it is safe to perform a dynamic modification. The dynamic modification functionality must consider the timing rule before it can perform a safe dynamic modification.

Constructing the integral section for a particular proposed modification is not a straightforward procedure. For example, as explained in Section 5.2, it is not possible to write a general-purpose algorithm that can construct the accurate integral section for a particular modification. Since every dynamic modification has its own integral section, the programmer must remain diligent. It is possible to develop procedures that have the

potential to help him or her in his or her efforts, but the procedure results tend to be only partly correct. Ultimately, the programmer has to decide the accurate integral section.

Once the programmer identifies this section and informs the dynamic modification functionality of it, the modification functionality then makes sure that grains are altered only when the execution pointer is outside the integral section. To ensure this, a communication mechanism should exist between the programmer and the underlying dynamic modification functionality. Once the system receives communication from the programmer, it should ensure that dynamic modification is performed accordingly. A good example of such a system is DYMOS [OUS 2], wherein the programmer specifies *when* conditions, along with the patches to update, as in

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

The above condition specifies that procedures `P` and `Q` should be updated only when procedures `P`, `M` and `S` are instantaneously unused grains (IUGs). When the programmer supplies the dynamic modification functionality with this kind of command, it initially checks to see if the corresponding grains can indeed be modified. If they can, modification immediately takes place. If, however, these conditions are not met, modification will be delayed until they can be. When execution resumes, the dynamic modification functionality monitors progress until said conditions are met; then the corresponding grains are modified. PODUS [OUS 1, 5] also falls within this category.

The advantage of this method is that it functions independently of the software architecture. The only requirement for the software architecture is that it be designed in a well-structured and modularized way, which describes virtually all modern software.

Despite its positive points, this mechanism possesses one key disadvantage: the underlying implementation of a technique based on this mechanism is complex. Most of the complexity arises because the underlying dynamic modification functionality must monitor the process until all validity conditions are met. Typically, if the underlying modification functionality realizes that the validity conditions the programmer supplies cannot be met, then it has to let the target process resume, but it continues monitoring the execution until conditions are met. This monitoring mechanism is complex, because it forces the executing process to temporarily relinquish control to the dynamic modification functionality, which can add overhead. Since identifying the integral section is not a straightforward process, a significant programming overhead can occur during the patch development.

Software Hot Swapping, the dynamic modification technique design that this author proposes in this thesis is based on implicit safe spot mechanism. The advantages of this method have been instrumental in making this decision.

5.2 Essential Components of the Integral Section

As noted earlier, whenever a grain needs to be modified, the programmer has to associate it with an integral section, which is a set of contiguous regions of the control flow graph of the program. Section 5.2.1 provides a description of the construction of the integral section for a single grain. Section 5.2.2 reports on the situations that can arise where integral sections of different grains must be merged, if a modification involves multiple grains.

5.2.1 Integral Section for a Single Grain

Essentially, the integral section of a grain depends on its scope and functionality. Section 5.2.1.1concentrates on describing the relationship between the scope of the grain and the integral section. Section 5.2.1.2 provides a description of the relationship between the grain functionality and the integral section. Section 5.3 reports on the details of a procedure that describes the construction of the integral section.

5.2.1.1 Grain Scope

The first issue that affects the integral section is the scope of the grain, because this limitation defines the lifetime of the grain's identifiers. Since every grain has an embedded functionality, it becomes highly probable that said functionality cannot be compatible with that of the new grain version when dynamic modification occurs. Moreover, if the execution pointer executes half a grain, there are two ways by which grains can be modified. The first is to let the execution pointer exit the grain, and the second is to let execution roll back. Grain roll back is a difficult task that requires much more effort on the part of the programmer. Because of overhead associated with roll back, it is better to let the grain complete execution. Hence, the dynamic modification functionality must modify a grain only when the execution pointer is outside its scope.

5.2.1.2 Grain Functionality

The functionality of a grain depends on the algorithm that defines it. An algorithm is a well-ordered collection of unambiguous and effectively computable operations that, when executed, produces a result within a finite amount of time. When the programmer implements an algorithm as a program, its underlying operations have to be implemented in the form of grains. Hence, grains interact with each other, as the algorithm dictates, and generate the net result that the algorithm is meant to produce. Whenever the execution pointer enters a grain, it is possible to infer that it reaches that grain after having passed through a number of other ones that comprise the underlying algorithm. The combination of all possible paths along which the execution pointer could pass before it reaches the target grain is called the *control flow path* of the respective grain.

It is possible for the functionality of a target grain to depend on other grains present within its control flow path. Dependency refers to the impact the encapsulating algorithm has over a target grain and its control flow path, as discussed below. The target grain is dependent on its control flow graph if the control flows from the target grain to its

predecessors, which occurs when they are its antecedents. This happens if the target grain is embedded within an *iterative grain*, which makes the execution pointer iterate over its scope. `for` and `while` loops can be termed *iterative block grains*. If a function calls itself using recursion, it can be called an *iterative function grain*. Similarly, if there is a circular recursion involving a set of functions, the collective set of all those functions is termed an *iterative program unit grain*.

The nature of the dependency depends on the manner in which the algorithm is implemented. In general, grains constitute either of two possible functionalities: I/O or data processing. I/O refers to interaction between the grain and the external environment for the sake of input and output, while processing contributes to the data within the program. Based on these functionalities, it is possible for two kinds of dependencies to occur between a target grain and its control flow path: *external dependency* and *data flow dependency*. The following subsections provide discussions of these dependencies.

5.2.1.2.1 External Dependency

In general, the programmer designs algorithms so that they interact with the outside environment in meaningful ways. If the target grain has I/O functionality within it, there is every possibility that it is within an external interaction mechanism in which the process interacts with the external environment in a certain way that only the external environment can understand.

For example, if a program takes in ten integers and outputs the average of those numbers, it interacts with the external environment during two separate instances: first, when it inputs the ten numbers and then when it outputs the average. The first instance is an iterative interaction mechanism with a meaning only the user understands. The output is just coupled with the input and the processing. However, if this entire mechanism, which consists of inputting ten numbers, processing the average, and outputting the average, is encapsulated within a function and if that function is called iteratively, it is possible for it to constitute an even larger iterative interactive mechanism. Programs can possess any number of such mechanisms. In fact, programs can possess nested iterative interactive mechanisms whose underlying semantic meaning only the programmer can characterize.

A target grain is said to be externally dependent on its control flow path if

- The target grain has I/O functionality within it, and
- If the target grain is part of a well defined interaction mechanism whose collective meaning only the external environment understands.

If the target grain has I/O functionality, it is necessary to identify the encapsulating interaction mechanism first. Then the programmer must examine the effect of a dynamic modification on that interaction mechanism, because, depending on the location of the execution pointer within the interaction mechanism at the instant he or she initiates dynamic modification, any number of effects can occur. After the examination, the

integral section for the target grain has to be identified. Only the programmer can decide the size of the integral section, because he or she alone can understand the interaction mechanism and its effects with respect to the target grain. Since interaction mechanisms are part of the functionality of the program, this external dependency can be termed abstract.

Figure 5.2, which describes a paint problem, demonstrates the above argument. The `for` loop represents the iterative interaction mechanism and its functionality is to paint the screen with red. The `Display_line_on_screen` function is responsible for drawing each line and the `for` loop is responsible for calling the `Display_line_on_screen` function. In this case, only the programmer can ultimately decide the size of the integral section, because it is not possible to write an algorithm to detect such dependencies.

Figure 5.5 represents the example shown in Figure 5.2. Here the target grain is externally dependent on the encapsulating `for` loop. The dotted arrow in Fig. 5.5 represents the abstract external dependency.

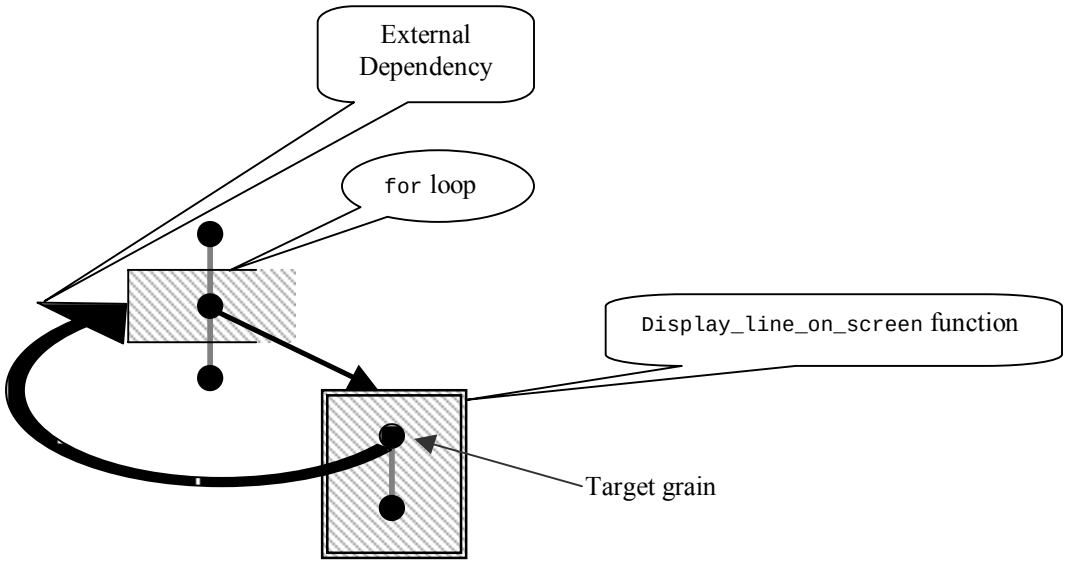


Figure 5.5: Pictorial Representation of Paint Problem

5.2.1.2.2 Data Flow Dependency

In general, it is possible to attribute data flow within a program to the incremental interactions of identifiers across the control flow graph. Every grain larger than that of an assembly language statement can have identifiers declared within its body scope, the lifetime of which terminates at the end of the corresponding scope. Based on this fact, this section gives an informal description of the data flow graph. This data flow graph becomes the basis for presenting arguments related to validity assurance. Formalizing these validity arguments is left for future work.

In order to assign data to an identifier, the programmer initially allocates some memory to it. The functionality which allocates memory resides within grains. The assignment

operator helps the programmer associate data with the respective identifier. Here the data flow into the location associated with the identifier from one of three possible sources: constants, other identifiers, or the input. Once data flow into the location associated with an identifier, the identifier can be used for three possible operations: re-initialization of the identifier with new data (which can occur any number of times), an output operation, or as influence on the value of a different identifier. This third operation can be termed *calculation* or *usage*. For example, in the statement

```
a := b + c
```

the calculation of the values of "b" and "c" influences the value of "a." Here, data flows from "b" and "c" to "a."

Given this logic, it can be said that the data flow graph consists of two kinds of nodes: *definition* and *usage*. Definition nodes represent the points where an identifier is initialized with data. Usage nodes represent the points where an identifier is used.

As noted above, data originate from input, constants or other identifiers. For a formal definition, let **S** represent the set that constitutes nodes of data input and the assignment of constants (definition nodes). Usage nodes represent the points where the identifier is involved in an output or a calculation, if the identifier is on the right hand side (RHS) of the assignment operator. If the identifier is present on both the left hand side (LHS) and the RHS, then that node is treated as a usage node, which can also contain initialization or cause re-initialization as influenced by other identifiers. This means that data flow from the influencing identifier(s) to that identifier to which datum will be assigned.

Relational expressions in this context are also considered usages. Let **C** represent the set of usage nodes. With these definitions, the data flow graph can be formally defined as follows:

A data flow graph $G = (V , A)$ consists of a set of nodes $V \in S \cup C$ and a set of edges $A \subset (S \times C) \cup (C \times C)$, where **S** represents the definition nodes and **C** represents the usage nodes.

Data originates at the definition nodes, and then they flow into other usage nodes. Finally they converge to nodes where output operations occur. For example, Figure 5.6 shows the data flow graph for the function `fun1()`. Aho, Sethi, and Ullman describe similar data and dependency flow graphs [EX 19].

In Figure 5.6, the data flow originates from lines 3 and 5. Data first flow from line 3 to line 4, a definition node where value is assigned to the location identified by `GlobalVar`. Again, line 6 receives data from lines 4 and 5. After usage, the data go to line 7. The gray boxes in Figure 5.6 represent the definition nodes. Double lined arrows represent the data flow. The nodes encapsulated within solid lined white boxes represent the usage nodes.

```
void fun1(void)
{
1     int GlobalVar;
2     int tempVar;
3     tempVar = 10;
4     GlobalVar = fun2(tempVar);
5     TempVar = 20;
6     GlobalVar = fun3(GlobalVar, TempVar);
7     Printf("%d", GlobalVar);
}
```

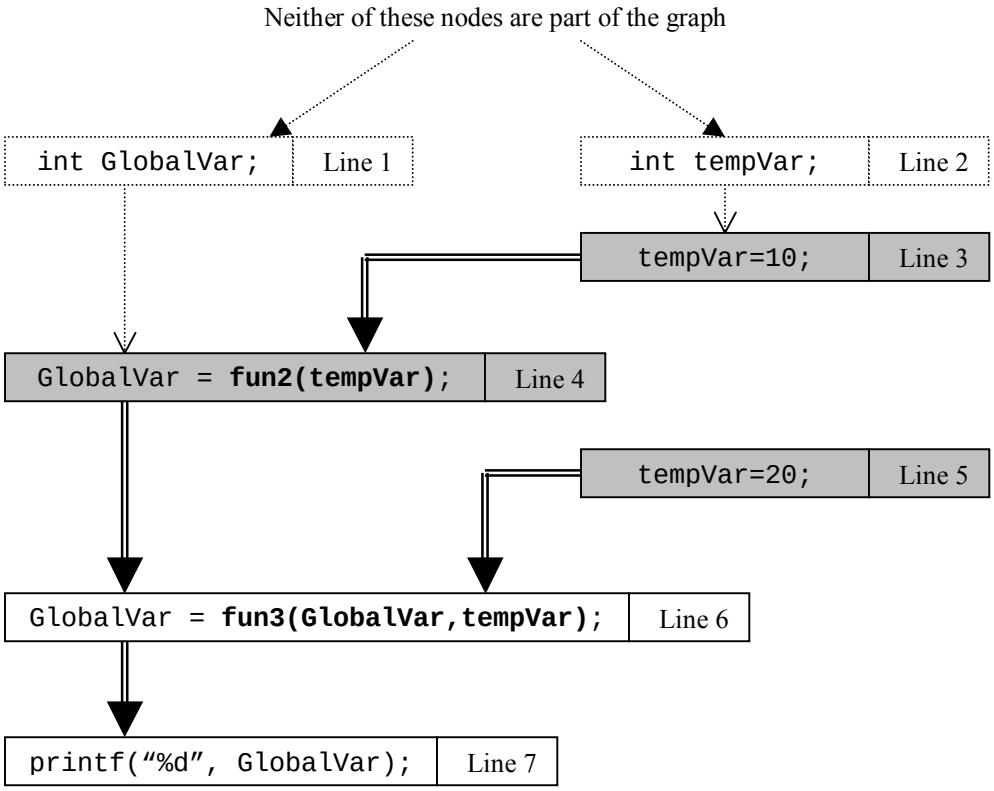


Figure 5.6: Sample Data Flow Graph

This data flow graph can also contain cycles, which are created whenever certain data must be iteratively calculated and can vary in size depending upon the implementation. Data flow is said to be in a cycle if data circulate within an iterative grain; their flow indicates an accumulation operation. Data cycles depend on the control flow. For example, the presence of data in a cycle need not mean that the corresponding identifier is within iteration, because data can flow from one identifier to another. Since data are computed throughout its lifetime, the functionality of the program is said to be under computation. Most data computation intensive programs have extensive data flow within their scopes.

In general, a program can have multiple disjoint data flow graphs completely disconnected from each other. For example, some could exist solely for controlling

selection and iteration. Likewise, the programmer might want to embed a program with multiple functionalities, each one of which can be assigned an individual data flow graph.

Once a program with multiple data flow graphs begins, the execution pointer reaches the grains that contain the nodes of these graphs. Since visualizing this entire process in a sequential manner is a complicated task, this author proposes a special *semantic graph* unique to every program. This semantic graph can be constructed by mapping all the data flow graphs of a program onto the corresponding control flow graph. This graph is similar to the dependency graph Johnson and Pingali propose [EX 18].

Since every grain of a program resides within the corresponding control flow path, it is possible to argue that every grain within a program likewise resides in this semantic graph. However, it is not mandatory for a grain to have nodes that reside in any of the corresponding data flow graphs, although in some cases, a grain can contain nodes of a number of disconnected data flow graphs of the same program. Given this analogy, data flow through a grain only if that grain contains at least one node of any of the data flow graphs.

When the programmer wants to modify a grain, if that grain does not contain any of the data flow nodes, then modifying the grain will not lead to internal semantic incompatibility. For example, modification of a statement from `printf("Hello");` to `printf("Hello World");` does not create any internal semantic incompatibility, because these statements do not contain definition or usage nodes. However, if the grain contains a node that is within a data flow cycle, a greater possibility exists for such incompatibility. A data flow cycle is nothing more than a data accumulation operation. Whenever one of the nodes in a data accumulation operation is modified when the execution pointer is computing the corresponding accumulation operation, a potential internal incompatibility can occur. In this case, the integral section must begin at the point where the data flow cycle starts, and end at the point where the data flow cycle stops. The same concept can be extended to a grain with multiple nodes of different data flow graphs.

5.2.3 Integral Section for Multiple Grains

If multiple grains require dynamic modification, the integral section concept must be reexamined, as it is in this section.

If the programmer wishes to dynamically modify multiple target grains of a program, situations can arise wherein some of them depend on each other. This situation can occur as a result of either data flow or external dependency. If there is a dependency between any of the target grains, then the corresponding individual integral sections might not prevent potential incompatibilities. The following example illustrates this issue.

Figure 5.7 represents three functions of an initial version, called Version 1, with the data flow graph of `fun1()` shown (See Figure 5.6). During execution of Version 1, the following steps take place:

1. `fun1()[version 1]` calls `fun2()[version 1]` and passes a value 10.
2. `fun2()[version 1]` computes formal argument and returns the value 100.
3. This 100 is stored in the location represented by the identifier `GlobalVar` in `fun1()[version 1].`
4. `fun1()[version 1]` then calls `fun3()[version 1]` and passes the identifier `GlobalVar` and the value 20.
5. `fun3()[version 1]` computes formal argument and returns the value 120.
6. This value is then stored at a location identified by `GlobalVar.`
7. Finally, 120 is displayed on the screen.

```
void fun1(void)                                    int fun2(int dVar)
{                                                  {
1     int GlobalVar;                                     return dVar*dVar;
2     int tempVar;                                 }
3     tempVar = 10;
4     GlobalVar = fun2(tempVar);                   int fun3(int gVar, tVar)
5     TempVar = 20;                                {
6     GlobalVar = fun3(GlobalVar, TempVar);              return dVar + tVar;
7     Printf("%d", GlobalVar);                     }
}
```

Figure 5.7: Version 1

Now, when that the programmer modifies the above version of `fun2()` and `fun3(),` Version 2, shown in Figure 5.8, is the result.

```
      int fun2(int dVar)              int fun3(int gVar, tVar)
      {                               {
        return dVar*dVar+10;            return dVar + tVar -10;
      }                               }
```

Figure 5.8: Version 2

When the execution pointer is at statement 5 (Figure 5.7) within the `fun1()` function, the variable `GlobalVar` points to a location that contains a value of 100. At this point, if both `fun2()[version 1]` and `fun3()[version 1]` are modified to `fun2()[version 2]` and `fun3()[version 2]`, as soon as execution resumes, a value of 100 is passed to the function call `fun3()[version 2]`. `fun3()[version 2]` returns a value of 110, which is unacceptable.

If `fun1()` calls the Version 1 functions, a result of 120 is displayed. Similarly, if `fun1()`calls the Version 2 functions, the same result of 120 is displayed. But if `fun1()`calls one function of Version 1 and the other function of Version 2, a problem arises. Figure 5.9 represents these scenarios.

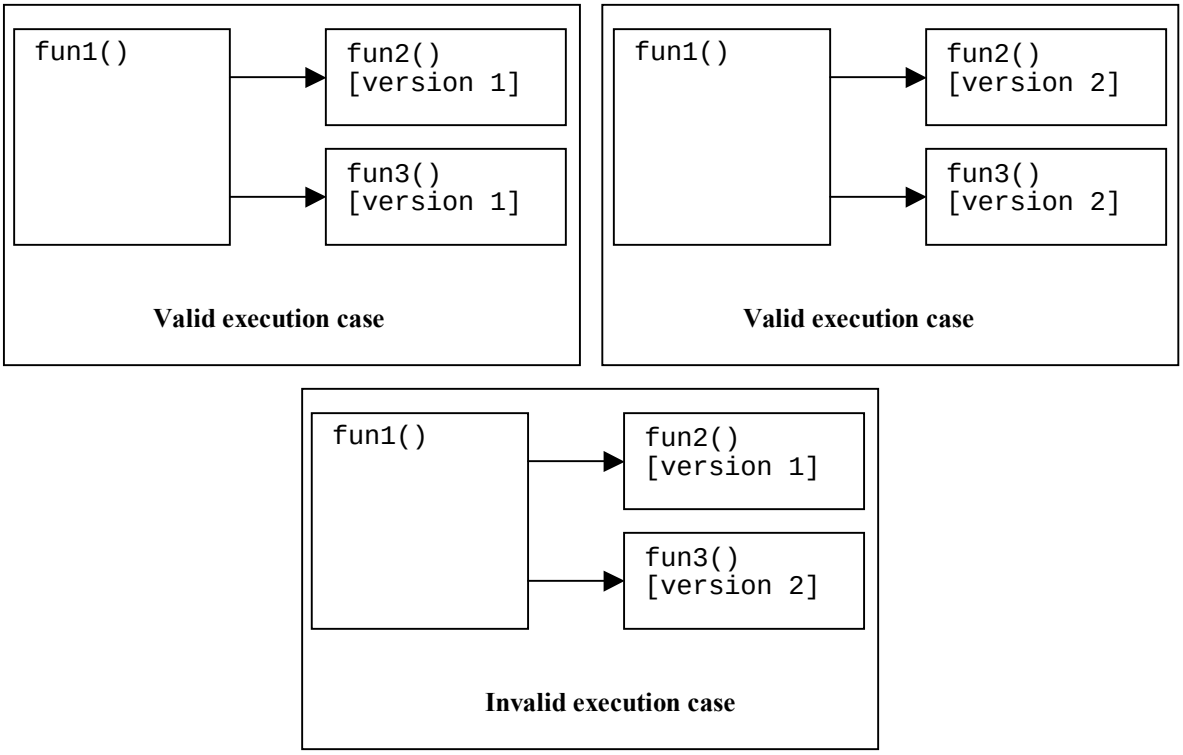


Figure 5.9: Valid and Invalid Cases

The main problem represented here is that a data flow path occurs from `fun2()`to `fun3()`(line 4 to line 6 of `fun1()` in Figure 5.6). Data flow from `fun2()` to `fun3()`. The dynamic modification functionality performs the update in the middle of the data flow and hence incompatibility occurs. The integral sections of both the functions `fun2()` and `fun3()` are their respective scopes. It is possible to prevent this incompatibility if `fun2()` and `fun3()` are modified at the same time, before the execution pointer reaches line 4 or after it crosses line 6 (Figure 5.7).

In order to avoid this internal semantic incompatibility, modification of target grains that have data flow across each other must take place at the same time, when these data are not flowing. Essentially the integral sections of these dependent grains need to be integrated to form a *super integral section*, which should also include the path along which data flow between the grains. The target-dependent grains should be modified when the execution pointer is outside this super integral section. In the context of a valid dynamic modification, this requirement is essential.

Multiple target grains can have external dependencies that are similar to the data flow dependencies explained here. Only the programmer can identify such external dependencies and incompatibilities and then construct the appropriate integral sections. However, Section 5.3 provides an explanation of an automated algorithm that can detect target grain data flow dependencies within a semantic graph.

5.3 Integral Semantic Algorithm

Gupta proves that it is impossible to develop a general-purpose algorithm to decide the precise time at which dynamic modification can be performed, simply because no such algorithm can detect all potential incompatible situations for a given set of target grains [OUS 6, 7]. Hence, it is not possible to design an algorithm that can ensure a 100% valid dynamic modification for a given set of target grains.

This author, however, argues that behind construction of an integral section lies a combination of decidable and heuristic procedures. Although, as Gupta points out, developing a general-purpose algorithm to perform the undecidable procedures is theoretically not possible, this author argues that it is possible to package decidable procedures in the form of an algorithm. Since constructing the integral section for every target grain involves tediously examining different graphs, this author proposes a framework wherein the programmer takes care of all undecidable procedures, and an *integral semantic algorithm* assists him with the decidable procedures. This section concentrates on describing the reasoning behind such a procedure and finally presents the proposed framework.

As has been noted, potential type incompatibilities can be detected using an algorithm; developing timing rules for the dynamic modification, however, is only partly computable. Whenever the type of an identifier is modified, either of two things can happen: the programmer either (a) modifies the type of all the other interacting identifiers or (b) casts the type to all interactions of the corresponding identifier. The integral section now can algorithmically be deduced as the lifetime of the modified identifier. If the types of the other identifiers are also changed, then the integral sections of all those identifiers have to be combined to form the aforementioned super integral section. Hence, valid dynamic modification of all identifiers can occur when the execution pointer is outside this super integral section.

However, this solution might not be suitable when the programmer modifies the type associated with a global variable. Since the programmer allocates the memory for a global variable within the data region, modification of this variable is much easier than modification of a stack-based variable declared within a function, because it is possible to reallocate the memory for the global variable at a different place within the data area, and to update all the corresponding address references of that global variable. If the variable is within a stack and if its type is modified, all the words that are between the modified variable and the stack pointer have to move around (due to the current machine instruction sets e.g., x86). This is true if the programmer modifies the identifier from an integer to a float. This modification is much more complicated than global variable modification. Although not theoretical, this is an implementation issue.

The programmer must determine a safe spot in such a way that it is possible to modify the type of the global variable. The programmer must also simultaneously modify the value of the identifier. If a global integer type is modified to float, the contents stored within the associated memory must be changed as well. If type definitions are modified,

the underlying issues tend to become complicated. All of these issues are further described in Chapter 6, Section 6.2. Comparing the type information of the modified version stored in the symbol table during compilation with the type information of the previous one can detect potential type incompatibilities. In order to avoid potential type incompatibilities, the programmer must be involved in developing the timing rules accordingly.

Examining the semantic graph provides a way to detect internal semantic incompatibilities. Whenever nodes that are within data accumulation operations must be modified, the integral section for such nodes encapsulates the respective iterative accumulation operations. Detecting these iterative accumulative operations by computational means is a straightforward process. Parsing the data flow from the nodes of the target grain can help in locating the accumulative procedures. It is possible to automatically develop timing rules in such a way that a target grain is modified only when the execution pointer resides outside the respective integral section. Ultimately, however, the programmer must examine these automatically generated timing rules and assure validity.

Detecting potential external semantic incompatibilities is difficult. The process can interact with a number of different kinds of external environments, all of which have their own way of interpreting the external state. It is, therefore, very difficult to develop an algorithm that can detect all potential semantic incompatibilities based on interpretations of the interacting environments, as described previously. Hence, again it becomes the responsibility of the programmer to detect the potential external incompatibilities and construct a suitable integral section, which then makes generating timing rules simple.

Unfortunately, all of the above arguments lead to a semiautomatic dynamic modification validity assurance system, where the best an automatic algorithm can do is detect potential incompatible situations and make best-guess recommendations that the programmer must examine and optimize based on his or her requirements. The detection algorithm cannot fully discover potential external inconsistencies. However, a combination of both an automatic algorithm and the programmer can make dynamic modification simpler and valid. While the algorithm is not 100% accurate, it significantly reduces the programmer's effort in the validity assurance process. Some burden still remains, but unfortunately, there is no other way to ensure 100% validity. This author argues that this kind of algorithm is acceptable as long as it can detect the potential type and internal incompatibilities accurately.

To an extent, the algorithm can detect potential type incompatibilities and generate rules to ensure a valid dynamic modification. This algorithm can also detect potential internal semantic incompatibilities and generate the corresponding rules. It is possible for these rules not to ensure efficient safe spots and integral sections, but they reduce the programmer's effort significantly. Since detecting internal semantic incompatibilities can be a laborious burden for the programmer, the fact that this algorithm can detect such incompatibilities simplifies his or her task. Finally, only the programmer can detect external semantic incompatibilities and avoid them by developing the appropriate rules.

Most of the arguments made within this section are based on reasonable arguments. Formalizing these arguments is left for future work.

The algorithm that generates the integral sections for the target grains is called the *integral semantic algorithm*. If the programmer modifies the initial version to a final version, the algorithm performs the following steps:

1. Constructs the control flow graph of the initial version of the program by parsing and analyzing the source code. Parses the source code of the initial version and generates all the possible data flow graphs. Maps the generated data flow graphs over the control flow graph, thereby generating the semantic graph.

2. Locates all the target grains within the initial version that have been modified to generate the final version.

3. Selects any two grains that have not been selected together earlier. Determines if there is data flow from one target grain to another, and, if there is, integrates the integral sections of both grains. The data flow path and the control flow path between both those grains also must be added to the integrated integral section (super integral section). The collective set of all these grains is comparatively larger. The significance of this collective set is that any dynamic modification of the corresponding grains remains valid as long as the execution pointer is not present within the corresponding integral section.

4. Generates the integral section for the identified dependency and lets the programmer verify it. The programmer should be able to edit the collective sets and the integral sections. In fact, he or she should be able to add new integral sections, modify the existing ones, identify new dependencies and paths, define super integral sections and the corresponding grains, and so forth. The programmer has to identify the external dependencies and construct the required integral sections.

5. Goes to Step 3 until the all combinations of target grains and their dependencies are examined and integral sections constructed.

6. Generates the timing rules to determine when the grains can be modified. The theory behind these timing rules is that the dynamic modification functionality must be able to identify the integral section for each grain or a set of grains and ensure that the grains are modified when the execution pointer is outside the associated integral sections. The timing rule must contain enough relevant information about the integral section and the corresponding target grain/s.

7. Permits the programmer to verify the generated timing rules and lets him or her edit the timing rules or manually generate new one, as needed.

8. Generates and deploys a patch file containing the binary differences (new grains) and the timing rules, as soon as the programmer finalizes the timing.

Chapter 6: Process of Safe Modification

6. Introduction

The dynamic modification functionality performs memory changes that are carefully timed to avoid potential incompatibilities, but must modify target grains only when their associated timing rules are satisfied. Hence, during the execution of the target process, the dynamic modification functionality has to monitor the state of the execution and perform modifications at instants when the associated timing rules are satisfied. Such an action leads to a conceptual execution scenario (*pseudo concurrent dual functionality*) where the dynamic modification functionality simultaneously monitors and modifies the executing target process.

This chapter is divided into two important sections. Section 6.1 provides a description of the concept of pseudo concurrent dual functionality. Section 6.2 reports on the procedures used to perform type modifications. Although Chapter 3, Section 3.4 contains a brief description of type modifications, Section 6.2 concentrates on the finer details of the procedures used to perform type modification.

6.1 Pseudo Concurrent Dual Functionality Concept

As described above, every dynamic modification system contains two functionalities: the target executing and the dynamic modification functionality. The target functionality constitutes a long running task that serves a predetermined purpose. In contrast, whenever a dynamic update is necessary, the dynamic modification functionality monitors the target functionality and performs modifications at appropriate instants of time, thereby ensuring validity.

DYMOS provides a good example to demonstrate this argument. In DYMOS, the target executing functionality is completely independent of the dynamic modification functionality. The dynamic modification functionality initially inputs the object code of the modified function (or a group of functions) along with the timing rule. It then monitors the state of the executing target functionality and waits until the associated timing rule is satisfied. As soon as this happens, the dynamic modification mechanism dynamically modifies the target process.

Although the target functionality is conceptually independent of the dynamic modification functionality, clearly distinguishing both these functionalities in some of the existing techniques is problematic. For example, dynamic classes method requires the programmer to design the software architecture in such a way that each class gets instantiated frequently. The underlying class loaders make sure that class instantiations are used as the de-facto safe spots for the associated class modifications. From an abstract perspective, it is possible to classify the class loading mechanism as a primitive form of monitoring mechanism, which waits for the execution pointer to reach a safe spot (class instantiation event) and then dynamically modifies the respective class. Here, the

dynamic modification functionality is implemented partly within the class loader mechanism with the rest within the software architecture. Popcorn also seems to have the same characteristic.

A clear distinction between the target and the monitoring functionality is always advantageous in the sense that it is possible to achieve separation of concerns. While developing his or her long running application, the programmer can concentrate exclusively on trying to solve the target problem instead of concentrating on the software architecture design that permits hot swapping. This is the reason why this author believes that methods like DYMOS are more flexible than the class loaders mechanism. Irrespective of the flexibility aspect, every dynamic modification method must have some form of the dynamic modification functionality that monitors and modifies the target functionality, thereby ensuring validity.

6.2 Type Modifications

The three kinds of type modifications described in Chapter 3, Section 3.4 are:

- Modification of stack based data objects
- Modification of data and heap based data objects
- Modification of function interfaces

This section is divided into three sub sections. Section 6.2.1 reports on the issues involved in modifying stack based data objects. Section 6.2.2 provides a description of issues involved in modifying heap and data based variables. Finally, Section 6.2.3 concentrates on the issues involved in modifying the interface of a function.

6.2.1 Stack Based Data Objects

The lifetime of their encapsulating functions limits the lifetimes of stack based data objects, and all instances of stack based identifier type codes are likewise present therein. The type code required for accessing stack-based data objects varies based on hardware architecture and design of the code generator. Once the compiler generates the type code, it remains constant through out the lifetime of the program. Moreover, such optimizations as peep hole and global have the potential of blurring the boundaries of the type code.

Based on such factors, the resulting instances of type codes of the same identifier that access the associated stack based data object can vary drastically. Now, if a stack based data object must be modified, there is every possibility that all instances of the type code related to the object likewise must be modified, which creates overhead. Moreover, the stack is implemented around the *Last In First Out* model, wherein the executable instructions within the text area can access the stack based data objects by using relative addressing techniques. If a data object is modified within the stack, there is every possibility that the delicate relative addressing might be disturbed.

Figure 6.1 provides a useful example for examining this issue:

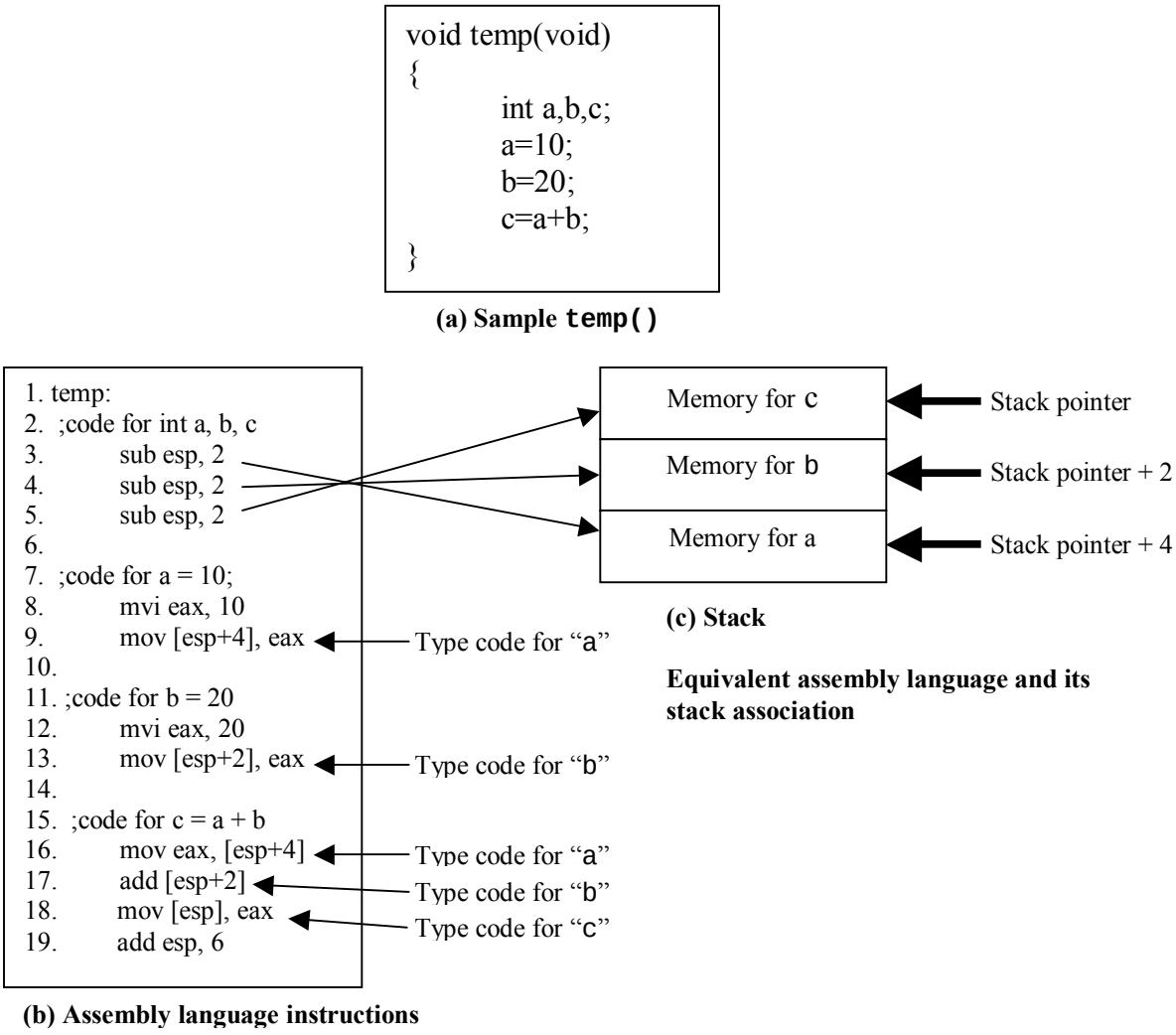


Figure 6.1: Relative Addressing Example

Figure 6.1(a) shows a simple function `temp()`. Figure 6.1(b) shows that function compiled into an assembly equivalent. Lines 3-5 in Fig 6.1(b) show the compilation of the source code statement `int a,b,c;` into three grains, wherein two bytes of storage are allocated for each integer. Essentially, the three "`sub esp,2`" instructions push up the stack pointer by six bytes, thereby allocating memory for the three integer variables. Computational operations start once the execution pointer crosses the fifth line. Line 9 can access the data object that corresponds to `a` by adding 4 to the contents of the stack pointer and accessing the resulting memory address. The same steps are used if it is necessary to access `b`.

Assume the dynamic modification process wants to modify the type of `b` from a 16-bit integer to a 32-bit integer. If this process extends the size of `b` by moving the data object of `c`, it needs to update the type codes that access `a` and `b` with respect to the new relative location. If the stack is further extended (above `c`), and the data object of `b` is copied to

the new extension, then the type codes that access `a`, `b` and `c` must be updated again. This implies that for any data object modification, the type codes of all data objects within the function likewise require alteration.

The location of data objects is calculated relative to the location identified by the stack pointer, the value of which changes whenever a new data object is added to the activation record. The relative addressing instructions are generated statically during compilation, which can be a problem of this relative addressing problem. An alternative involves calculating the address of data objects relative to the beginning of the activation record. In general, the beginning of the activation record remains static throughout the life of the function. If complicated optimizations are considered, the situation quickly turns chaotic, because it is possible for the boundaries of the type code to become blurred.

Based on the above analysis, this author argues that whenever a stack based data object requires modification, it is simpler to modify the function as a whole. Whenever a stack based object is modified, situations might arise where offsets corresponding to other objects within the same function, particularly those involving the stack pointer, also need alteration. The only way to avoid this situation is to modify the encapsulating function when the execution pointer is outside its scope, an approach that restricts type modifications within the main function. Since modifying objects in the data or the heap area is much simpler, one solution involves inserting all data objects of the main function into these locations rather than into the stack.

6.2.2 Data and Heap Based Data Objects

In general, data objects stored in the data and heap areas are associated with absolute addresses obtained when the program is loaded into memory. For global variables, relative addressing techniques can also be used to calculate these absolute addresses, but unlike the case of stack based objects, here the memory location relative to which the relative address is calculated remains static. In most cases, the relative memory location can be either (a) the offsets within a global offset table (GOT) [SL 13] or (b) statically derived offsets. Similarly, absolute addresses of heap based variables are obtained whenever data objects are explicitly created. In general, heap based data objects are created during the runtime and hence, statically derived offset philosophy does not apply in their case.

In most cases, memory pointers contain absolute addresses and are associated with uniform type codes. Every pointer must contain the absolute address or a mechanism which calculates the absolute address. The type code of general stack based identifiers is more efficient than that of pointer type codes, because the former uses machine instructions that are specially designed for enhancing the execution efficiency. For example, depending on the hardware features, the statement "`a:=b`" can be translated into

```
Mov eax,[:::] // move the contents of b onto the accumulator
Mov [:::],eax // move the contents of the accumulator into the
                 memory allocated for a
```

or

```
Mov [:::],[:::] // move the contents of a to b directly
```

Some hardware architectures (for example, x86) do not support the `Mov [:::],[:::]` kind of instructions. In general, most systems possess a wide range of machine instructions for performing basic arithmetic operations involving memory registers and the stack. The x86-based architecture is designed around this concept. Yet there are drawbacks: the stack based type code is not uniform, and the pointer based type codes do not seem to carry many efficient machine instructions (due to the absolute addressing capability). The use of available machine instructions can, however, make the stack based type code efficient. Based on this author's experience, most of the widely used code generators are designed this way.

Since the type code of general pointers is uniform, it is possible to update the data object and corresponding instances of the code by simply copying the former to a different memory location and updating the latter accordingly. Again, this is where the "safe spot" comes into play. For example, if the programmer wants to expand an array during an update, he or she can specify a safe spot where the size of the array can be expanded. If that array is at the end of the data area, expanding the size of the array does not create any memory overwrites. If another data structure follows the array, the whole array needs to be copied to different free memory and then expanded. Once the array is moved to a different memory location, all the type codes that refer to the array must be updated immediately. In this situation, modifying the type code should be a straightforward process.

If the programmer wants to modify a linked list into a binary tree, he or she again must first choose a safe spot. Once that is identified, the programmer must write an *Execute Once and Throw Away* (EOTA), or a convert routine, code to create the binary tree and copy data from the nodes of the link list to the tree's nodes. The dynamic modification process executes the EOTA code while it simultaneously modifies all the instances of the type code of the link list in such a way that they now access the binary tree. With this approach, data structures can be modified as well.

6.2.3 Modifying the Interface of a Function

In general, functions are associated with input and output types; in fact, a function itself is a type. A function, in general, can take in arguments and return a value. The programmer develops a function in such a way that it assumes control whenever the execution pointer enters the scope of the associated function calls. During compilation, the compiler converts these function calls into respective instances of type codes. In general, type codes of function calls can be classified into four categories.

The first category of type codes, the *V-V Case* (void-void), does not take in arguments and does not return values. The type code for its function call contains a `call` instruction that reaches the corresponding function. This `call` instruction stores the address of the next instruction in the stack and then directs the execution pointer to target function. Whenever the address associated with the instruction counter is within the domain of the

`return` instruction, the address stored within the stack is popped off and the execution pointer takes it on. Figure 6.2 shows the assembly type code for this kind of a function.

Case	Function type	Equivalent Assembly Type Code
V-V case	`temp()`	`call temp`

A-V case	`temp(a)`	`push [esp+offset1]`
		`call temp`
		`add esp,2`

V-R case	`a=temp()`	`sub esp,2`
		`call temp`
		`mov eax,esp`
		`mov [esp+offset1],eax`
		`add esp,2`

A-R case	`a=temp(b)`	`push [esp+offset2]`
		`sub esp,2`
		`call temp`
		`mov eax,esp`
		`mov [esp+offset1],eax`
		`add esp,4`

Figure 6.2: Function Calls and Equivalent Assembly Type Codes

The second category of type codes corresponds to function calls that involve argument passing without any return value and is, therefore, known as the *A-V Case* (argument-void). Here, the argument is pushed onto the stack before a `call` instruction is placed. Figure 6.2 shows the assembly type code for this function call. Here, the argument is 16 bits long.

The third category of type codes, the *V-R Case* (void-return), corresponds to function calls that return values without passing any arguments. Here, sufficient space is allocated on the stack to hold the return data object before placing a function call. The called function moves the return value onto the allocated stack space and then places a `return` instruction (See Figure 6.2). In this case, the return value is 16 bits long.

The fourth category of type codes, called the *A-R Case* (argument-return), corresponds to function calls that involve both return values and argument passing. Sufficient space is allocated on the stack to hold the return data object before placing a function call. The argument is also pushed onto the stack. The called function moves the return value onto the allocated stack space and then executes the `return` instruction. Figure 6.2 shows the assembly type code for this kind of a function call, and both argument and return values are 16 bits long.

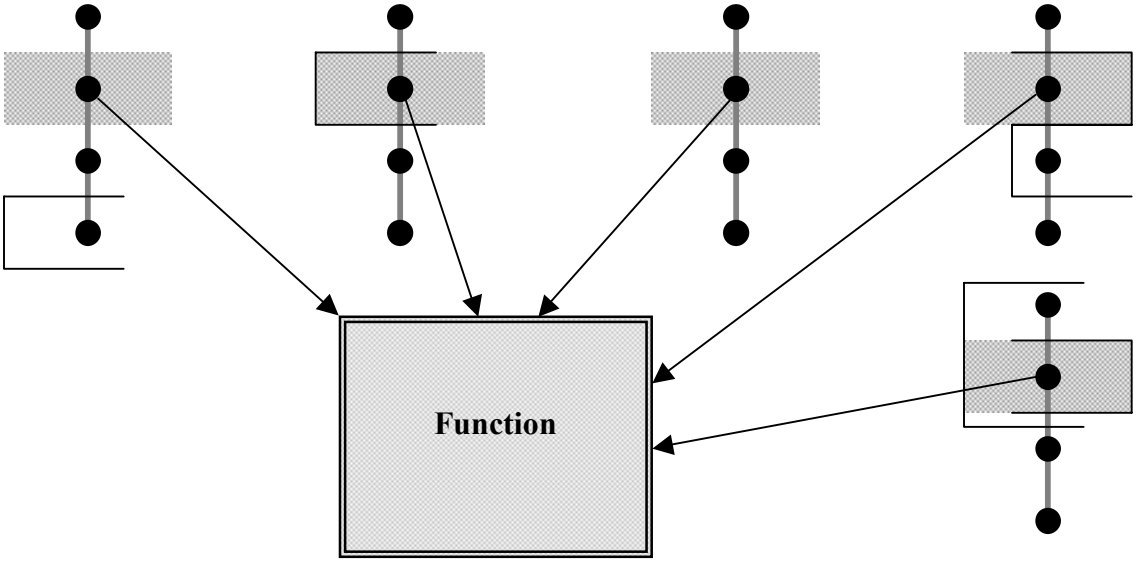


Figure 6.3: Integral Section for the Modification of the Interface of a Function

Now, if the type codes shown in Figure 6.2 are examined, it is possible to see that all type code categories, except for the *V-V Case*, insert stack based data objects onto the stack before passing control to the required function. Since stack based data objects are involved, this author argues that the integral section for this kind of modification should encompass the scope of the target function, as well as all the instances of the type code that correspond to that function.

Essentially, because of the existence of stack based data objects, the dynamic modification functionality must not modify the interface of the function as long as the execution pointer is present within the scope of the integral section. Since stack based objects require type code that uses relative addressing, a function's interface must not be modified until the execution pointer is at a safe spot relative to the integral section containing the functions and its type code.

Chapter 7: Implementation Considerations

7. Introduction

Before developing a dynamic modification technique, the software designer must address the basic characteristics and the underlying implementation issues typical of dynamic modification techniques. While the code of an executing process remains static throughout the lifetime of the process, the state constantly changes, depending upon the location associated with the execution pointer, the previous state, and the external inputs. Irrespective of the dynamic nature of the state, the executable code is always compatible with the state, because the latter is generated by the former with the help of user input. Therefore, the dynamic modification process must alter this static executable code in a valid manner.

Section 7.1 concentrates on characterizing dynamic modification techniques based on their nature. Whenever a software engineer (or a programmer) designs a dynamic modification technique, he or she addresses certain implementation issues, termed *design trade-offs*, which determine the complexity of the technique. Section 7.2 provides a discussion of these issues.

7.1 Components and Characteristics of Dynamic Modification Techniques

In general, dynamic modification techniques possess an associated set of characteristics that determine their usability. Segal and Frieder [OUS 1] propose seven characteristics, which should:

- Preserve program correctness
- Minimize human intervention
- Support low-level program changes
- Support code restructuring
- Update distributed programs
- Not require special hardware
- Not constrain the language and environment

The research in this study identifies eight general characteristics, which are different from those Segal and Frieder propose, and which include:

1. Transition Period
2. Control
3. Validity Assurance,
4. Process Model
5. Granularity of Dynamism
6. Platform Independence
7. Software Architecture
8. Perenniality

The following eight subsections provide descriptions of each of these characteristics.

7.1.1 Transition Period

Every dynamic modification technique requires a dynamic modification functionality that monitors and modifies the executing process based on the corresponding source code modifications and the timing rules.

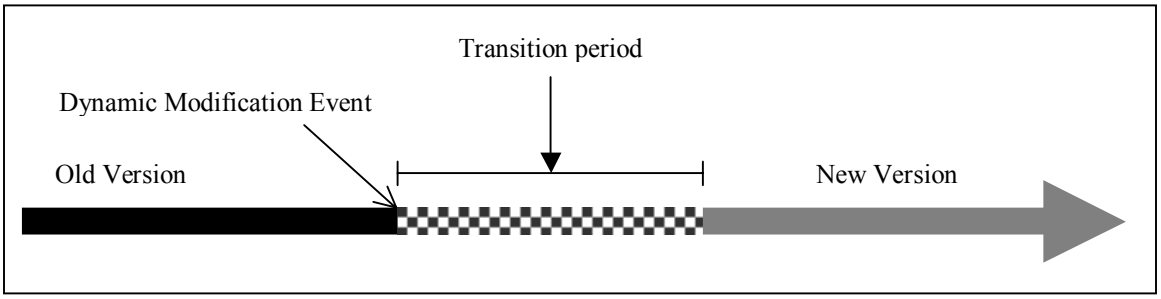


Figure 7.1: General Dynamic Modification

This functionality uses timing rules in the modification process. The event at which this modification functionality suspends the target functionality is called the *dynamic modification event*. The dynamic modification functionality takes a certain amount of time to perform the modifications. This time can include the waiting time as well as the time required to overwrite the old code with the new code. Figure 7.1 shows the *transition period*, which reflects the time period between the modification event and the instant at which the modification is said to have completed.

It is usually difficult to predict the transition period, because it depends on the location of the execution pointer with respect to the dynamic modification event. If the event occurs when the pointer is within the safe region, the transition period remains small. If, however, the pointer is within the integral section, the dynamic modification must delay modification until it crosses over to the safe section. The same arguments hold for the explicit safe spot mechanism, as explained in Chapter 5, Section 5.1.1. In general, it is desirable to have a small transition period, because its end marks completion of the modification. The software engineer must design every dynamic modification technique in such a way that the transition period is comparatively small. If it is large, then users are deprived of the latest modifications and service until the transition is complete, which is undesirable.

7.1.2 Control

The control characteristic deals with an authoritative entity that has the right to dynamically modify a particular target process. If, for example, there is a process executing on a particular hardware system, the question of who has the authority to modify this process arises.

Software Hot Swapping fits within a vendor-client scenario. In this scenario, the vendor develops an application and ships it to the client, who starts using this application. However, whenever the vendor develops a new patch, the question of who is going to initiate the dynamic modification arises. Vendor can certainly initiate the dynamic modification, but, in the case of enterprise software, users might want to upgrade software when they want to instead of being at the mercy of the vendor. In this author's experience, most Chief Information Officers (CIOs) do not want to upgrade their mission critical software unless they are certain the new version will be more economical. There are certain applications (Set top boxes) where the vendor wants to initiate the dynamic modification. This is an important characteristic of a dynamic modification technique.

7.1.3 Validity Assurance

Plattner and Nievergelt consider an executing program a moving point in an n-dimensional space [EX 12]. When dynamic modification occurs, it can be assumed that the moving point suddenly changes its movement and orientation.

Imagine that a pilot flying a Light Combat Aircraft (LCA) at the speed of 1.7 Mach wants to change the direction of his flight, either suddenly or sequentially. If the pilot makes a sudden change, the LCA has to be able to cope with many physical factors, including inertia and centrifugal forces, for the pilot to maintain control over it. The degree to which these physical factors affect the plane are directly proportional to the angle at which the aircraft has to turn and to the rate of change. If the degree to which the pilot turns the LCA is 180^{o} within a time of 10 seconds, then the results of inertia can be disastrous. If, however, the pilot turns the same angle within 60 seconds, the transition takes place smoothly with every physical factor under the pilot's control.

A similar situation applies to an executing program. When a program is executing, its functionality has an associated state. If the functionality suddenly changes (at an inappropriate instant of time) due to dynamic modification, incompatibilities arise which can cause a system crash. Generally, chaos occurs when the modified functionality is not compatible with the state of the executing process. The only solution for this problem is to control dynamic modification, as explained in Chapters 4 and 5. Every dynamic modification technique must ensure validity. The nature of the timing rules also depends on the aggressiveness of the version change.

7.1.4 Process Model

Each technique designed to perform dynamic modification is associated with an underlying process model, an abstract term called *kinetics* which is similar in effect to the DNA of a molecule. Like the DNA, which shapes and structures the organism, kinetics determines the results of each dynamic modification technique. The process model (*kinetics*) used to design the technique decides its practical usage and efficiency.

For example, in the case of DYMOS [OUS 2], Lee ensures that the same executing process is updated whenever modification is required. Gupta uses a different model in

which the programmer suspends the target executing process and initiates a new process [OUS 6, 7]. As soon as the new process is created, the dynamic modification functionality transfers data present in the suspended process to it. After the transfer is complete, the suspended process halts and the new one begins at that point. Similarly, the kinetics used in server farms can be termed *modification using redundancy*.

Whenever a dynamic modification event occurs, the state of the program can be non-determinate unless the functionality and the external inputs are known. The events that occur during the transition period are instrumental in modifying the functionality of the executing process. The software engineer, who designs the dynamic modification technique, has to design the process model of any modification technique around this aspect: events that happen during the transition period. While developing different mechanisms around these events, the software engineer eventually ends up developing a technique.

7.1.5 Granularity

Grains are the language constructs programmers use to encapsulate the functionality of algorithms. Granularity expresses the smallest grain that it is possible to modify using that particular technology.

Some modification techniques are limited to program units. In general, the existing object-oriented languages use classes, in which functionality is encapsulated. Each class has a well-defined name and can be considered a program unit equivalent to an imperative construct. As a class is of a higher granularity, it is possible to develop program architecture so that whenever a class needs to be instantiated, the underlying system explicitly looks for its latest version. The dynamic C++ class and the Java dynamic classes exemplify this technique[SL 7, 8]. However, this technique is not suitable for an imperative language, because program units do not have an associated name and are not instantiated. DYMOS, PODUS, and Popcorn can, however, modify functions. Granularity is an important characteristic of a dynamic modification technique.

7.1.6 Platform Independence

The process model of any technique should be largely independent of both the underlying operating system and the programming language within which the modification technique is implemented. DYMOS, PODUS, and Popcorn are relatively independent of the underlying operating system and the programming environment.

7.1.7 Software Architecture

As the example in Chapter 4, Section 4.1 shows, if a program is modified when the execution pointer is within the corresponding integral section, it is possible for an invalid state to occur. To solve this problem of timing, the programmer must locate safe spots for every possible dynamic modification. In this identification process, he or she is apt to use some of the tools, such as the integral semantic algorithm. The aggressiveness behind this

principle varies from design to design, but the software architecture has a dramatic effect on the construction of the integral section.

For example, the dynamic C++ class technique and the Java dynamic classes require that programs be designed in such a way that every class gets instantiated frequently [SL 7, 8]. Thus, whenever a class requires modification, the pointer to the new version must be given to the executing process; as a result, then the corresponding class needs to be instantiated, the new version is used. This requires a specially designed architecture. Dynamic modification techniques that are based on implicit safe spot mechanism do not require any specialized software architectures, because the target functionality and the modification functionality are implemented as two distinct responsibilities.

7.1.8 Perenniality

Perenniality is an important characteristic for dynamic modification techniques. When a process is dynamically modified, the dynamic modification functionality uses its own algorithms to modify the machine instructions within the address space. Suppose that a grain has been dynamically modified and that the algorithms (of the modification functionality) have also altered the contents of the address space. Once modification is complete, the programmer might need to re-modify the grain to a newer version, something the dynamic modification functionality should easily accomplish. The ability of the technique to permit repeated modification is called *perenniality*.

Perenniality can be examined from two perspectives: the machine code domain and the source code domain. A technique is said to be perennial only if it has both machine instruction and source code perenniality. From the perspective of the machine code domain, when a program is compiled, the compiler generates machine code in a pattern relative to the source code. When modification occurs, the associated algorithms of the dynamic modification functionality modify the instructions of the program's address space. It is possible for this grain modification to disturb the code pattern, which can make future modifications difficult. The dynamic modification functionality must take considerable care to ensure that a modified grain can be repeatedly modified; this is known as *machine instruction perenniality*.

From the second perspective, the source code domain, when the programmer modifies the interface of a function, he or she must design the program in such a way that the function works with a new set of argument and return types. Since a function interface change accompanies modification to the statements that call the respective function, all the previous researchers consider it wise to modify the function alone and use *ad-hoc* convert routines to manage the type modifications. Here, the old function calls pass arguments of the original types. The *ad-hoc* convert routine converts these values from the old type to the new type that the modified function accepts. After performing the type change, the *ad-hoc* method passes those values to the new function. The new function computes and returns the result. If required, the *ad-hoc* method modifies the type of the return value and passes the control to the calling statements.

There are, however, several drawbacks to this procedure. Not only is it complicated, but it also becomes increasingly difficult to keep track of the convert routines as the interface of a function is modified a few times. Moreover, these routines restrict the degree to which the new function interface can change, compared to the original interface. Finally, the insertion of *ad-hoc* code causes the technology to change the compilation patterns in the address space, and over time that grain becomes increasingly difficult to modify. All the existing methods that permit interface modifications follow this method, even though it has these flaws. DYMOS, PODUS, and Popcorn are good examples that employ this mechanism.

Figure 7.2 helps explain this issue more clearly. Here, the function `tempFun` contains a call to the function `dumpFun`. Assume that there are many other functions within the same program that place similar calls to `dumpFun`.

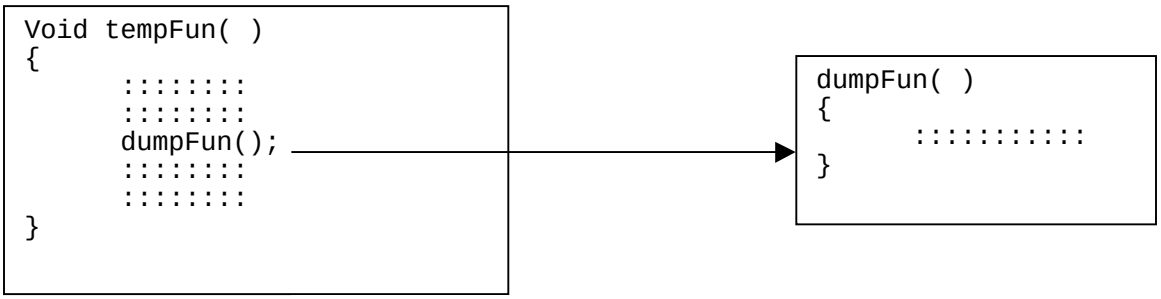


Figure 7.2: Original Definition of `dumpFun( )`

Now, assume that the programmer wants to change the interface of `dumpFun` in such a way that `dumpFun` takes in an integer parameter, as Figure 7.3 shows.

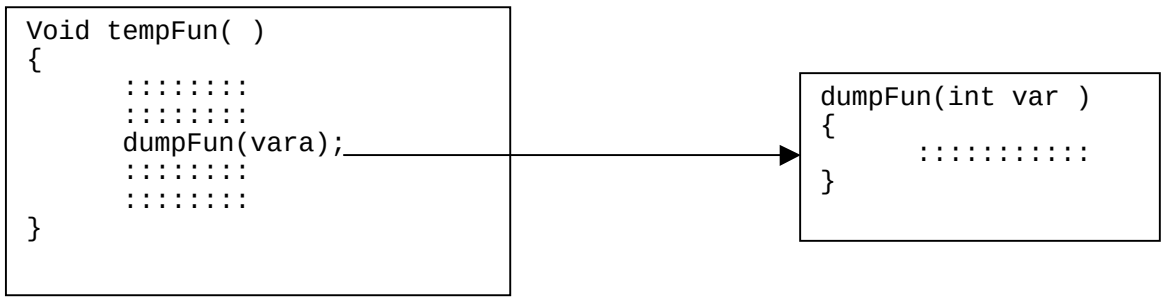


Figure 7.3: New Definition of `dumpFun( )`

Implementing this modification involves altering `dumpFun` and all functions that call `dumpFun`. When an attempt is made to dynamically modify the old version of `dumpFun` with the new one, all the functions that call `dumpFun` must, in turn, be changed. Segal and Frieder, Lee, and Hicks all consider modifying the calling functions a complex process and instead propose an intermediate piece of code, the stub, which actually gets called in place of `dumpFun` [OUS 1, 2, 9].

Figure 7.4 shows that this stub converts the old interface of `dumpFun` to the new one. Here, stub `stubFun()` converts the interface of `dumpFun` from void to an integer. With this method, the programmer does not need to modify all functions that call `dumpFun`. From the perspective of software architecture, this method is not flexible, because the true function call mechanism is not contemplated during implementation. This restricts the extensibility of the program.

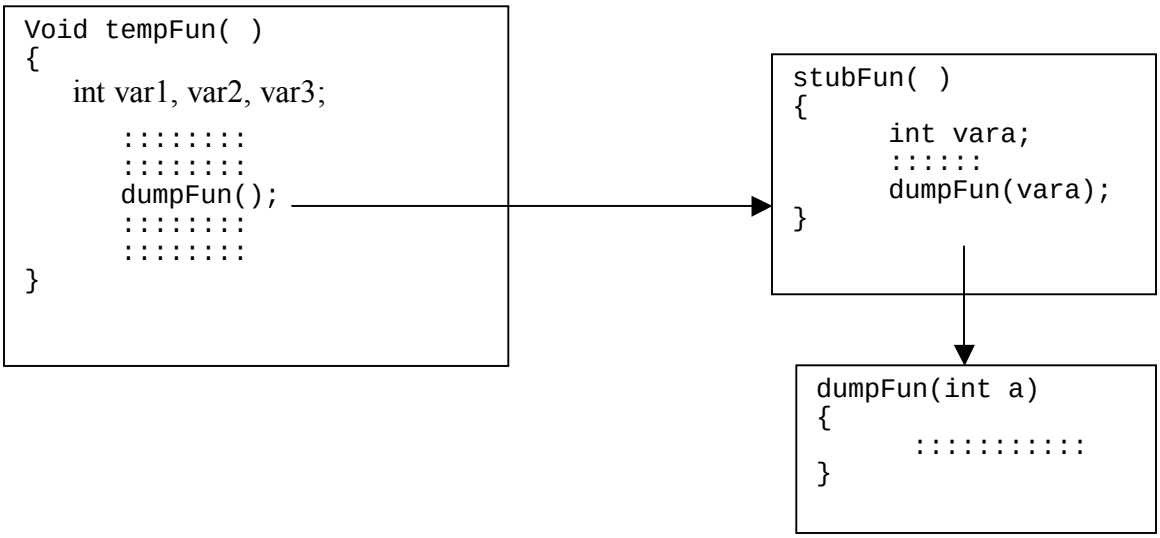


Figure 7.4: Implementation of the New Definition of `dumpFun()`

7.2 Criteria and Design Issues

Dynamic modification techniques have a set of implementation trade-offs that further decide their complexity. This section examines the issues dynamic modification raises, including overhead. Section 7.2.1 provides an explanation of the problem of programmer perception, while Section 7.2.2 concentrates on detailing some of the relevant technical complexities.

7.2.1 Programmer's Perception

The first overhead cost involves the programmer's perception of the entire computational process involved in dynamic modification. For example, during the transition period, it is much easier for the programmer to visualize a single large metaphorical operation than a set of smaller ones. Likewise, it is easier for him or her to keep track of functions that are being modified than it is to remember all of the blocks and statements of each function.

Since most people tend to avoid "too much detail" scenarios, the programmer will have more success identifying functionality by an abstract metaphor than by a set of minute metaphors. For example, if statements are to be dynamically updated, then the encapsulating function must not be optimized when the program is initially compiled. The programmer has to keep this in mind before compilation. He or she has to guess which potential functions must not be optimized. Also, he or she has to ensure that he or

she does not optimize those predetermined functions during the next modification. This is very difficult, because it entails predicting grains for future modification.

However, as the grain size decreases, the programmer tends to move from an abstract functionality overview into a minute functionality scenario. If the programmer has to perform dynamic modification on a single small grain, he or she faces no problems keeping track of what happens during the process. However, if he or she has to dynamically modify different granular levels simultaneously, he or she tends to keep track of modifications that are at much higher granular levels, simply because it is easier to remember a *program unit change*.

If the programmer loses his or her perspective, human error can result, which places the entire executing process in jeopardy of crashing. Because of the potential problem of programmer perception, this author argues that any dynamic modification technology should be associated with a good programmer interface that helps keep track of the applied dynamism on an abstract basis. The complexity of this programmer interface tends to increase as the size of the grain decreases.

7.2.2 Technology Requirement

Like the programmer's perception, the required technology also contributes to the complexity of dynamically modifying grains irrespective of their size. A discussion of the five problems that have to be solved follows in Sections 7.2.2.1-7.2.2.5.

7.2.2.1 Grain Mapping Information

To dynamically modify a grain, the dynamic modification functionality must first locate it within the executing process. Every grain is located at a specific location and is associated with both the starting and ending addresses of the process. These source code grains must be mapped in relation to their corresponding object code grains. This *grain mapping information* is at a minimum when granularity is within the function and program unit regions of the grain hierarchy. As the dynamic modification functionality moves down the hierarchy, the number of grains and the grain locality information increase.

For example, in a program that contains five program units with each program unit containing six functions, the memory required to hold the grain mapping information of one grain is **L** bytes. The grain mapping information necessary for all the program units is **5 * L** bytes. If the grain information of all the functions is also considered, then a total of **5 * L + 5 * 6 * L** bytes will be the amount of memory required to store all the grain mapping information. The same memory size argument can be extended to smaller grains.

It can then be stated that as grain size decreases, the amount of grain mapping information needed increases. The exact space required to hold this grain mapping information depends on the kind of storage algorithm used, but the amounts of grain mapping information and required computational power, in general, increase as grain size

decreases. If statements are to be modified, then enormous grain mapping information is required. Optimization creates another hurdle, as Section 7.2.2.5 explains.

7.2.2.2 Process Monitoring Mechanism

The subject of *safe spots* raises two underlying issues. First, the dynamic modification functionality must know the safe spot for the corresponding grain, known as the *programmer-system communication mechanism*. Secondly, the dynamic modification functionality has to observe the process whenever dynamic modification is required. Dynamic modification of a grain can be performed only when the modification functionality realizes that the associated timing rule has been satisfied.

The programmer-system communication mechanism constitutes a communication path between the programmer and the dynamic modification functionality, wherein the former relays information regarding the timing rules to the latter. The latter then monitors the executing process, modifying it only when the associated timing rule is satisfied. If the timing rule is never satisfied or if there are any other errors, the monitoring functionality alerts the programmer, who acts accordingly.

The dynamic C++ class technique and the Java dynamic classes follow the explicit safe spot mechanism [SL 7, 8]. Here, a class is dynamically modified whenever the corresponding old class requires instantiation. Moreover, the programmer does not need to inform the monitoring functionality, in both cases, the dynamic class loaders, about the safe spots. In Hicks' method [OUS 8, 9], as well as in DYMOS, the programmer has to inform the modification system.

Since this process monitoring mechanism ensures that modification occurs only when the timing rules are satisfied, implementing it is the single most important issue in the design of a dynamic modification technique.

7.2.2.3 Garbage Collection Overhead

Garbage collection creates overhead, because it can be a complex process. For example, if one small efficient grain replaces three old grains, some space remains. Likewise, if a large grain needs to replace a small grain, it needs to be copied to a different location and a jump pointer needs to be placed so that the new grain assumes control whenever the corresponding old grain is called. The old grain address area becomes unused space, which must be *garbage collected* in order that the memory can be used more efficiently. The size of the unused space depends on the size of the grain. As grain size decreases, the amount of reclaimable space also decreases. If that space is not reclaimed, memory fragmentation results, creating *islands of unused memory* within the text area of the process. Modifying the data can also create unused memory. If that space is to be reclaimed, there is another cost trade-off.

The space that can be reclaimed from a program unit is larger than that which can be reclaimed from a statement. The garbage collection algorithm requires some processing

for locating and reusing each of these islands of unused memory, and this processing is not without cost. The effectiveness of this collection depends on how many atoms can be filled within each of these islands. As grain size decreases, garbage collection becomes complicated and inefficient. For example, if grain size is on the order of 3 to 4 machine instructions, an attempt to reclaim that memory might not be worth the effort.

In this author's opinion, if all memory is reclaimed after dynamic modification, irrespective of grain size, the process can become very complicated and inefficient. The garbage collection algorithm must be efficient and less time consuming so that during reclamation the executing process remains suspended only for a small amount of time. While this does not adversely affect dynamic modification, it is important to the efficiency of the overall process.

7.2.2.4 Referential Environment

Within an imperative program, global variables are part of the referential environment of every grain. In general, good software engineering practice involves (a) declaring few global variables with lifetimes equal to that of the program and (b) many sets with lifetimes encapsulated within functions, blocks, and statements.

During execution, the number of global variables is constant. Whenever the execution pointer enters a function, the referential environment increases in two steps. The first step occurs when the execution pointer enters the function definition part, where function parameters are added to the reference environment. Once the instruction moves inside the function, step two occurs: a number of other local variables are declared and then added to the environment. Once the execution pointer comes out of the function, the referential environment decreases to the global variables and the return value of that function.

Compared to functions, blocks generally contain more declarations. The referential environment of a block constitutes all variables present within the encapsulating function's environment, plus variable declarations made inside the corresponding block. Hence, the referential environment of the block remains the same or increases with respect to the encapsulating function. The size of a statement's referential environment is going to remain the same, because statements generally do not contain explicit programmer declarations whose lifetimes end at the end of the scope of the statement. Therefore, based on the above arguments, Figure 7.5 illustrates that the referential environment increases as the granular size decreases.

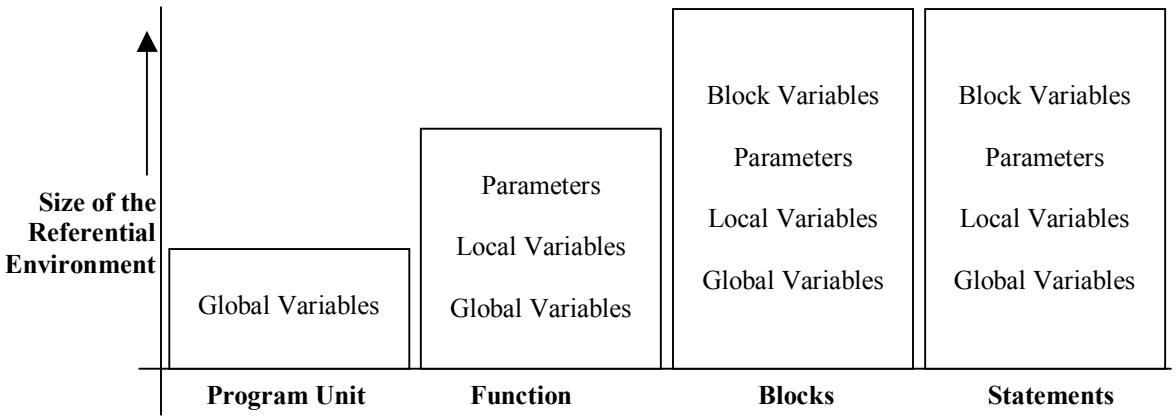


Figure 7.5: Size of the Referential Environment Vs Granular Size

Imperative languages usually do not have explicit declarations for program units, but languages like Pascal have *nested functions* that can access the variables declared within the encapsulating function. For the purpose of this discussion, these nested functions can be considered *named blocks*. In any case, the concept of increasing referential environment holds good for the nested functions as well. In general, the referential environment increases as the grain size decreases.

7.2.2.5 Compiler Code Generation

The code generation element of the compiler creates still more difficulties. The first difficulty with such generators is that they optimize the code after it has been generated. Since optimization is a unidirectional process, because once it occurs, it is not possible to distinguish the individual higher-level grains. For example, if the programmer wants to perform dynamic modification on grains that are smaller than functions, he or she should work to ensure that the compiler does not optimize across the boundaries of those grains. *Global* and *peephole* optimizations blur grain boundaries.

The way the compiler generates equivalent code for structures such as *if* and *while* structures, both of which can be considered blocks, raises the second difficulty. If the programmer converts the conditional statement *if* to a *while* loop without changing the body or condition of that block, then instruction compatibility issues arise. These issues depend on the architecture and the algorithms used to convert both *if* and *while* loops into corresponding machine instructions. The same argument can be extended to all smaller grains, such as statements.

It is possible to design a number of methods to bypass this problem. One method involves copying the complete grain at a different location and then putting a `call` instruction at the beginning of the old grain and a `return` instruction at the end of the new grain. This method works as long as the old grain occupies more words than the `call` instruction. Boundaries must not be blurred at any cost in this case. Regardless of the method, though, it must ensure *perenniality*. In any case, as grain size decreases, compatibility issues arise between old and new grains. Modifying a function is straightforward and can

be accomplished with a simple pointer relocation. Modifying a grain smaller than that of a function, however, causes boundary blurring and code incompatibility, which further increases the complexity of the required technology.

Chapter 8: Software Hot Swapping

8. Introduction

Software Hot Swapping is a type of dynamic modification that includes adding, changing, or deleting a program's responsibilities. The first section of this chapter reports on the specifications of this technique. Section 8.2 provides a discussion of the underlying functionality model. Finally, section 8.3 concentrates on explaining the program's implementation framework.

8.1 Characteristics of Software Hot Swapping

Software Hot Swapping is based on the implicit safe spot mechanism, which makes the technique software architecture independent. This is an important decision, because it is unrealistic for a vendor who might need to modify a program containing code of a million lines to adopt an architecture-dependent technique. The granularity used for Software Hot Swapping makes it possible to modify functions, blocks, and statements.

Software Hot Swapping is a technique that fits within a typical vendor-client scenario: a single vendor can have multiple clients, but every client has only one vendor. Hence, the vendor, the only one with control over a program's source code, is responsible for initiating dynamic modification in the user's software.

Compared to the DYMOS [OUS 2] model, Software Hot Swapping has a more advanced validity assurance system based on communication between the programmer and the underlying modification system. The programmer specifies the instances at which modification can take place, and the system follows these timing rules. For additional flexibility, though, the programmer can also perform advanced operations, such as the *Execute Once and Throw Away* code, which Chapter 9 reports on in detail.

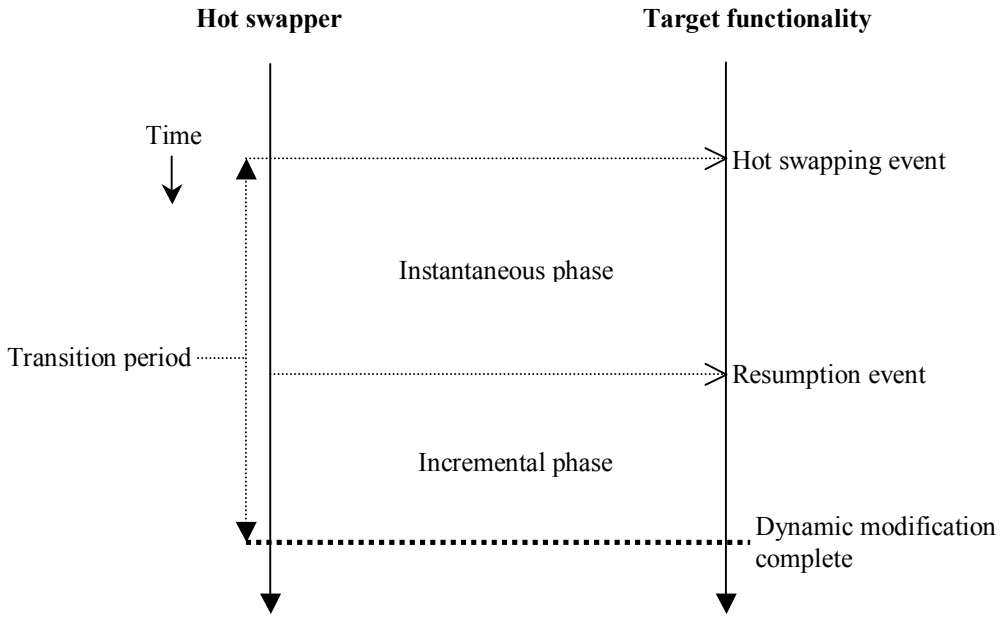
One of the main goals of this technique is high perenniality, which is the careful modification of the grains and the data objects in such a way that they can be modified repeatedly a number of times. However, this is an implementation issue rather than a theoretical issue.

8.2 Functionality Model

If the Software Hot Swapping concept is viewed from a different perspective, both the dynamic modification functionality(or the hot swapping functionality) and the target functionality can coexist (pseudo-concurrent dual functionality). Under this system, the hot swapping and executing functionalities are autonomous entities that execute like co-routines.

Here, the hot swapping functionality monitors and modifies the target executing functionality. When the user performs *hot swapping* (dynamic modification), the hot

swapper comes into existence and suspends the target system. Once this happens, the hot swapper checks to see if any of the timing rules can be satisfied. The hot swapper modifies its address space with the grains whose respective timing rules are satisfied. After this, the hot swapper lets the suspended target system resume execution and monitors the state of the execution. If any of the unsatisfied timing rules (with respect to the process suspension) is satisfied, the associated grains are modified and the target continues execution. The following paragraphs provides more details of this mechanism. This model clearly separates the hot swapping system from the target system (See Figure 8.1).



Hot Swapper	Target System
Instantaneous Phase	Process suspension
Incremental Phase	Interleaved execution

Figure 8.1: Software Hot Swapping Framework

Initially, the moment at which the hot swapper suspends the target functionality is called the *hot swapping event*. When this occurs, the hot swapper checks the timing rules to see if any grains can be installed. The target functionality is then immediately updated with all the modifiable grains (whose timing rules have been satisfied) during the *instantaneous phase*. Grains that cannot be modified during this phase are the *active grains* and the integral grains (See Chapter 2, Section 2.1.16). The hot swapper must

delay modification of these grains until their timing rules are satisfied. The hot swapper then lets the target functionality resume execution, the *resumption event*. After this, during the *incremental phase*, the hot swapper monitors the address space until any of the previously unsatisfied timing rules are satisfied. At any instant of time, if a timing rule is satisfied, the corresponding grains are modified. Once the hot swapper modifies all of the grains, it can be said that the process of dynamic modification is complete. The transition period in this model extends from the hot swapping event until the end of the incremental phase (See Figure 8.1).

The duration of the instantaneous phase is finite, which means it is equal to the amount of time required to modify all old grains. However, the duration of the incremental phase is difficult to calculate, because it is completely dependent on the nature of the timing rules and, to an extent, the software architecture. One mechanism to reduce this time is to efficiently modularize the entire software into small units, thereby avoiding a complete monolithic architecture based on the design Parnas suggests [SE 1, 2, 3]. In general, most existing software is designed using this concept.

Whenever a modification is needed, the hot swapper modifies the executing functionality by suspending it, altering it on the fly, and then permitting it to resume execution from the point of initial suspension. There is, however, a possibility of another approach wherein both functionalities can execute concurrently, in the sense both continue simultaneously. It is possible to visualize this as a two-processor system wherein the executing system continues on one processor, while the hot swapper executes on the other processor. The hot swapper modifies the executing process dynamically without suspending the target process. In this model, process suspension does not occur. This author argues that such an approach is suitable for time-critical applications.

There are many practical issues involved in this method, because both processes must cooperate with each other to ensure valid modification. Synchronization between them is also another challenge. This *Super Software Hot Swapping* is something more difficult to achieve and is beyond the scope of this research.

8.3 Framework

Since Software Hot Swapping is a flexible technology, it conforms to the typical vendor-client scenario in several ways. Figure 8.2 shows the framework consists of two mechanisms. The first, which takes place at the vendor's site, is the development of the *hot pack*, a file containing the difference in the binaries of the initial and final versions, as well as the timing rules and mechanisms by which modifications should occur. This first mechanism is briefly described below. Chapter 10 provides a detailed description of an implementation of the hot pack development environment. The second mechanism is the method by which the hot swapper uses the hot pack and updates the executing application at the client's site. The second mechanism incorporates two functionalities: the executing software application and the hot swapper, the functionality that performs hot swapping on the target application. Chapter 11, Section 11.1 provides a further description of the hot pack installation mechanism.

Figure 8.3 shows the hot pack development environment, which contains three essential components: the grain mapping component, the validity assurance component, and the versioning component.

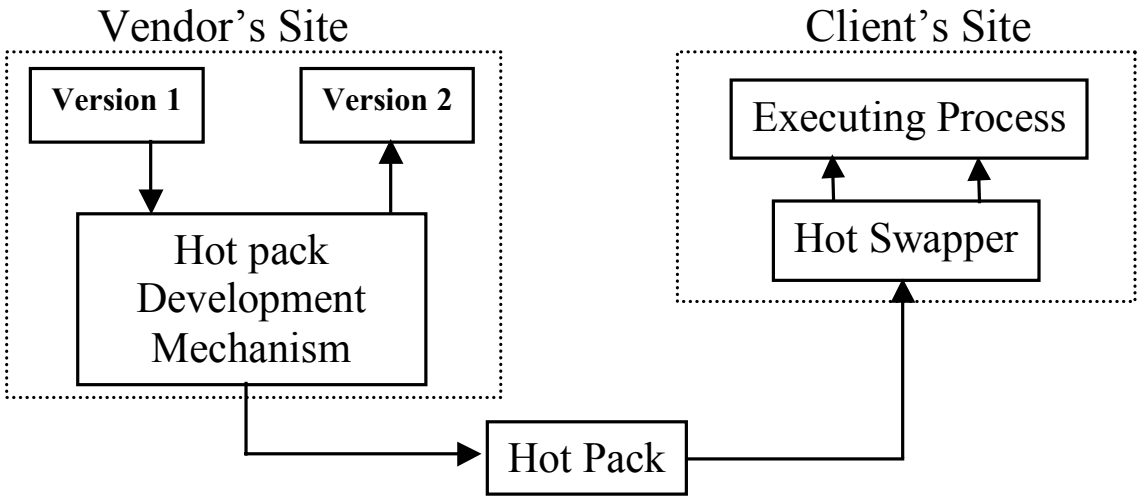


Figure 8.2: Framework

The first component is a module responsible for dividing the entire source code into grains and then mapping these grains with the corresponding object code grains during compilation. Whenever the programmer updates the source code, comparing the new grains with the grain mappings of the previous version can computationally localize the modified sections. Once this is accomplished, it is possible to compile the modified source grains into corresponding object code grains.

The second component, the validity assurance component, requires the programmer, after he or she completes the source code modifications, to identify the integral sections for the target grains. With the help of the validity assurance component, the programmer examines the control flow paths of all modified grains. Based on the control flow paths and the grain dependencies, timing rules that ensure a valid hot swap can be generated either automatically or manually. Hence, the programmer uses the second component to identify the integral sections and generate appropriate timing rules. It is also important, though, for this component to let the programmer develop his or her own *triggers* (Execute Once then Throw Away codes) that function at specific instances and offer flexibility, as Chapter 9, Section 9.1 concentrates on explaining.

The third component is the version manager, which acts as the central repository for grain mapping information and validity assurance rules for all versions of a software application. Such centralized control is crucial to keeping track of the program versions.

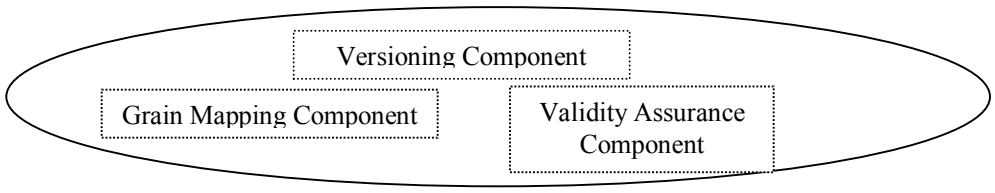


Figure 8.3: Components of a Hot Pack Development System

Chapter 9: Timing Communication Mechanism

9. Introduction

The hot swapper must ensure that every change is implemented only when the associated timing rule is satisfied. This author argues that there should be a mechanism by which the timing rule communicates the integral section along with the validity condition to the hot swapper. In some cases, the hot swapper may have to modify the grains when the execution pointer reaches a predetermined safe spot. This chapter provides a description of the informal semantics of timing rules. Formalizing the semantics is left for future work.

Section 9.1 reports on the concept of the triggers (Execute Once and Then Throw Away codes). Section 9.2 concentrates on describing execution roll back, while Section 9.3 provides a discussion of a communication methodology based on a special hot swapping calculus. The programmer uses this communication methodology to inform the hot swapper of when he or she intends to perform the associated modification. He or she accomplishes this step using timing rules.

9.1 Triggers

As noted previously, an old grain is modified into its corresponding new grain only when the location of the execution pointer and the state satisfies the corresponding timing rule. At this point, the programmer might want to execute a task that will change the state of the target process within the prior version to make it compatible with the new grains of the next version. To achieve this, the programmer can declare special trigger functions.

The programmer can use the hot pack development system to generate these triggers and any of the existing programming languages to develop them. The hot pack development system performs a syntactic check to make sure that the triggers do not contain any syntactic errors. The programmer can associate a trigger to a timing rule. Hence, whenever a corresponding timing rule is satisfied, the changes are implemented simultaneously with the execution of the trigger.

The hot swapper uses the triggers to change the state of a program or to invoke some of the existing functions within a program. Triggers should be able to access all data objects within the address space of a process, as well as all external library functions the process itself can access. Such triggers offer flexibility in controlling potential problems with incompatibility, as Section 9.3 concentrates on explaining in more detail.

9.2 Execution Roll Back

If the execution pointer is within the scope of a target grain during the instantaneous phase, it can be reset to the beginning of the scope and modification can occur. This method is called *execution roll back*.

Execution roll back is permitted only if the initial state of the process is restorable while the execution pointer is inside the scope of the grain. Typically, the process consists of the following three steps:

1. Restoration of the initial state,
2. Resetting of the execution pointer to the beginning of the grain, and
3. Modification of the target grain to a new grain, along with execution resumption at the beginning of the grain.

While the first step is debatable, the second and third steps are straightforward. The initial state can be restored if the target grain does not perform an operation involving modifying a non-stack based data object or interaction with the external environment. However, if the target grain involves either of these two operations, human intervention is required to revert the current state to the initial state. This effort is feasible only for coarse grains.

Of potential assistance in restoring the initial state during rollback is a trigger-containing code that records the state at which the execution pointer enters the scope of the target grain. For example, if the target grain modifies a global variable or a static variable, the corresponding trigger should document these changes.

In some cases, if the grain interacts with the external environment, it is not advisable to attempt rollback. For example, if the target grain passes a message to Internet location X saying that it is ready to receive data and X begins the transference, it is unadvisable to perform an execution roll back at this juncture. In fact, the programmer must identify this restriction before choosing the roll back option. While rollback is thus not always feasible, this author maintains that it remains an option.

9.3 Hot Swapping Calculus

Once the programmer, with the help of the validity assurance component, identifies any potential incompatibilities, he or she informs the hot swapper about how to modify the appropriate grains. The validity assurance component helps the programmer come up with safe and efficient instructions. This communication methodology constitutes the timing rules.

The performance of grain modification depends on the following:

1. Time at which modification is performed.
2. Trigger functionalities (if any).
3. Associated data object modifications (if any).
4. Associated modification of the function interfaces (if any).

All such modifications involve certain situation-specific validity assurance mechanisms, for which it is not possible to identify a general-purpose algorithmic solution, because the

computational process simply cannot understand the functionality of a program. The programmer must use a special hot swapping calculus to explain to the underlying system how such modifications must occur. While this section describes such a hot swapping calculus using informal arguments, formalizing of this method lies beyond the scope of this research.

When the programmer wants to perform hot swapping, he or she opens the executing program's source code in his or her editor, modifies it to a new version, and then compiles it. He or she then has to ensure that the underlying mapping system identifies all of the modified grains by comparing the old and new versions of the system and developing a list of differences between them. Every target grain has a corresponding new grain and a set or tuple containing a target grain. The associated new grain is called a *transition*. A collection of all possible transitions is a *Hot List*.

Once the Hot List is generated, the integral semantic algorithm attempts to identify potential incompatibilities, a process in which the programmer must be involved. Once incompatibilities are identified, the underlying system attempts to generate timing rules to avoid them based on the hot swapping calculus. Each timing rule is called a *dictum*. A list of all semantically ordered dictums for a particular dynamic modification is called a *dictatorial*, which is the programmer's depiction of how modifications must occur.

This dictatorial and the new grains are packaged in the form of a patch file called a hot pack, as noted previously. Once the hot swapper reads this hot pack, it uses the new grains within it, as well as the rules in the dictatorial, to perform hot swapping. Chapter 10 provides a detailed analysis of hot pack development.

The hot swapping calculus prescribes the semantic meaning of the dictatorial and helps the programmer construct rules. The hot swapper performs transitions based on these. Sections 9.3.1 and 9.3.2 provide descriptions of the properties of a transition and dictums, respectively. Finally, Section 9.3.3 reports on the construction of the dictatorial.

9.3.1 Transition

A transition is an instruction that incorporates two elements: the pointer to the target grain and the pointer to the corresponding new grain. For the purposes of this discussion, the target grain is called the *leftist* and the new grain, the *rightist*. A transition is said to have occurred if the rightist replaces the leftist present within the executing process.

A transition can contain a null leftist, which implies that a new grain is being added to the address space of the process. Similarly, whenever a leftist is supposed to be removed from an executing process, the rightist of the transition is represented as null. Each transition has an associated name that can be used in the dictatorial. A transition could also be associated with a trigger.

9.3.2 Dictums

Each transition is associated with a programmer-determined identifier, and each can be given a name generated by an automatic algorithm. In the following example, A is the identifier, while x1 is the leftist and x2 is the rightist.

```
A = (x1 , x2)
```

Every transition has its own dictum. Based on the constituent elements, a dictatorial set can be either a grain set or a transition set. Each set also has an associated identifier and is represented with a leading hash symbol (#). A set can look like this:

```
#tempset = (A, B, C, D);
```

Here **tempset** contains four elements: **A**, **B**, **C** and **D**. Sets can contain either grains or transitions, but not a mixture of both.

Each dictum can contain a number of transitions to which the associated transactions can be inter-related with Boolean operators, if an super integral region is involved. The cross operator (+) represents the Boolean operator **OR**. Similarly, the dot operator (•) represents the Boolean operator **AND**. **NOT** is represented by (**!**). The left and the right bracket operators indicate the precedence of the operations inside them. **IC** represents the execution pointer. A prefix **@** can represent a roll back. Each dictum has to be terminated with a semi-colon (**;**).

For example, a dictum of the form "<transition1> AND < transition2>" indicates that both transition 1 and transition 2 must occur at the same time. Section 9.3.3.3 provides further examples.

9.3.3 Dictatorial Construction

Since the hot swapper can perform hot swapping on a program at a remote location, the extent to which the programmer can express himself or herself depends on the calculus. In this author's opinion, a higher degree of expressiveness gives the programmer enough tools to perform whichever task he or she deems appropriate. Implementing the four control structures used in programming languages — sequence, selection, iteration, and parallelism — can enhance the expressiveness of the calculus. To further enhance it, the programmer can define sets that contain collections of transitions or collections of programmer specified grains, as described in the previous section.

9.3.3.1 Predefined Identifiers

This section contains some of the predefined identifiers essential during dictatorial construction.

IC represents the execution pointer. If it is within the scope of a grain, it can be represented as **IC → <grain name>**.

EG represents the set of all executing grains when hot swapping is to be attempted. If a grain is executing, it can be represented as **<grain name> ∈ EG**. This has to be read as the grain that belongs to **EG** set.

SG represents the set of all the suspended grains and the inactive integral grains.

IUG represents the set of all the Instantaneously Unused Grains (IUGs).

9.3.3.2 Dictatorial

Dictums can be executed sequentially or concurrently by using parallel constructs. Similarly, selection and iteration can be applied over dictums. The combination of the above described control structures enhances the expressiveness of the dictatorial.

A dictatorial can be constructed with dictum blocks, each of which contains a number of dictums. When the hot swapper executes all the dictums in a block, it is said to have executed. This occurs completely independently of the other blocks. To ensure valid concurrency, a transition present within one dictum block must not be present within another. A block can further contain any number of blocks within its scope. A set of blocks can be placed within a block by enclosing it with { and }. Each block begins with a left square bracket (**[**) and ends with a right square bracket (**]**).

The use of an **if** statement, which possesses a grammatical definition similar to that of the underlying language, can implement selection. The **if** statement is a construct within which dictums can be embedded. The syntax is as shown:

```
if (<condition>)
{ }
else
{ }
```

Similarly the **while** operator, the grammatical definition of which is similar to that of the underlying language, implements iteration:

```
while (<condition>)
{ }
```

During the process of hot swapping, the entire dictatorial is stored within the hot swapper at the client's site. During the incremental phase, this swapper gains control whenever a target grain relinquishes it. All blocks are executed concurrently; the dictums are executed sequentially.

9.3.3.3 Examples

This section concentrates on describing the construction of the dictatorial. The examples that follow represent different methods of construction. In each example, when all the operations in dictum construction are completed, the result is the dictatorial.

Dictum construction

If **A**, **B**, **C** represent three different transitions, and if the programmer wants to insert the rightist of **A** only if the transition of the rightist of **B** is complete within the address space of the executing process, this means that transition **A** depends on completion of transition **B**. This has to be read as **A** depends on the existence of **B**, which can be represented as

```
[
      A • (B);
]
```

Here the transition enclosed within the parenthesis has the highest precedence. The dictum can also be represented as

```
[
      B;
      A;
]
```

If **A** needs the existence of **B** and **B** needs the existence of **A**, which means both **A** and **B** have to be hot swapped at once, then both A and B have the same precedence. This is known as *interdependency* and has to be read as **A** interdepends on **B**. This can be extended to as many transitions as needed. The dictum for interdependency is

```
[
      A • B;
]
```

A situation wherein **C** depends on the existence of either **A** or **B** can be represented as

```
[
      C • ( A + B);
]
```

If a trigger must be associated with this dictum, it can be represented as

```
[
      (C • ( A + B)) exeTrigger( );
]
```

The above is a *post-event trigger*, executed immediately after the hot swapping expression **(C • (A + B))** is executed. If it is a *pre-event trigger*, it gets executed before **(C • (A + B))**. In that case, the hot swapping expression appears as

```
[
      exeTrigger( )(C • ( A + B));
]
```

Pre- or post-event triggers are distinguished based on the relative position of the trigger to the transition or dictum. Other triggers, known as *default triggers*, can be used within a dictum without being associated with any transitions; they execute automatically whenever the hot swapper passes control to them:

```
[
   exeTrigger();
]
```

Using triggers along with transitions, as shown below, can extend the expressiveness of this hot swapping calculus:

```
[
   (exeTrigger1() (C) • (A)exeTrigger2() ) exeTrigger();
]
```

Here **exeTrigger1()** occurs pre-event, which means before transition **C**. **ExeTrigger2()** is a post-event trigger, executed immediately after **A**. Also, **exeTrigger()** is post-event, occurring after the entire hot swapping expression is complete.

Integral Sections

The programmer creates a dictum in such a way that modification of corresponding grains is delayed until the execution pointer comes out of the integral section, which a specific name identifies as

```
[
      while ( IC → g1i ) {};
      A • B • C;
]
```

In the above example, $\mathbf{g1_i}$ represents the integral section. The **while** loop assumes control when the execution pointer comes out of the scope of general functions or at safe spots the programmer determines.

Selection

Dictums can also access data objects within the address space as the result of a selection statement the programmer designs:

```
[
      if (::counterValue == 10)
            A;
]
```

This effectively means that the transition **A** can occur only if the global value `counterValue` is equal to 10. To execute such a dictum, the hot swapper has to observe the `counterValue` whenever a grain modifies its value. Similarly a stack-based data object can also be used for selection purposes. For example, in the following example, **B** can occur only if the value of `counterClock,` embedded within the scope of `temp()` function, is equal to 12.

```
[
      if (temp::counterClock == 12)
            A;
]
```

Chapter 10: Integrated Development Environment

10. Introduction

This chapter provides a description of the details for a hot pack development mechanism in the form of an integrated development environment (IDE). This IDE possesses ten basic modules, which are described in Section 10.1. Implementation of such modules can differ, but their underlying mechanism is the same across different platforms. Sections 10.2 and 10.3 provide descriptions of the mechanism by which all ten modules interact with each other and generate the hot pack and the structure of the hot pack, respectively.

10.1 Components of the IDE

This section presents an abstract interaction of the ten modules within the IDE, which Figure 10.3 shows. The ten modules are:

1. Version Manager
2. Instantiation Mapper
3. Editor
4. Granulizer
5. Compiler
6. Hot Spotter
7. Hot Spotter User Interface
8. Dynamism Verifier
9. Dynamism User Interface
10. Hot Packer

The following ten subsections provide details about each module.

10.1.1 Version Manager

The version manager can be thought of as a *working over database* that contains detailed information about the program. The *first version* exists before all others. The *prior version* occurs before a subsequent version. The prior version can be, but is not necessarily always the first version. A prior version is always modified to a *next versio*n. The version manager profiles the following six categories of information needed for every version:

1. Version source code
2. Compiled object code
3. Grain mapping information
4. Optimization information
5. Hot list
6. Dictatorial

While the first three categories are self-explanatory, it bears noting that each version is associated with optimization information, which contains data regarding how each grain has been optimized during compilation. Since optimization generally disrupts mapping between the source and object codes, information relating to these problems is vital to future modifications. Information related to the hot list and the dictatorial for all versions except the first must be stored within the version.

The dictatorial of a next version describes how hot swapping must be performed on its corresponding prior version. This given, it is possible to generate a hot pack to swap the executing process of a certain version to any successive ones. For example, if successive versions 1, 2, and 3 exist, and if a client who is using an application (Version 1) wants to upgrade his or her system to the latest version, he or she can do this by using an appropriate hot pack that can modify Version 1 to Version 2 and Version 2 to Version 3.

10.1.2 Instantiation Mapper

The instantiation mapper module, which tracks the mapping between source code grains of the prior and next versions, is especially useful for developing a next version from the prior version. When this mapper is activated, the module copies the prior version from the version manager and creates a new instance by duplicating its source code. It then maps the grains of the duplicated copy, which is the next version, with those from the prior version. The editor then presents this new instance of the source code to the programmer.

During the course of modifying the source code for this next version, the programmer can add more grains, delete some, or modify those already in existence. However, whenever a modification is performed, the instantaneous mapper must ensure that the grain mapping is valid (See Figure 10.1).

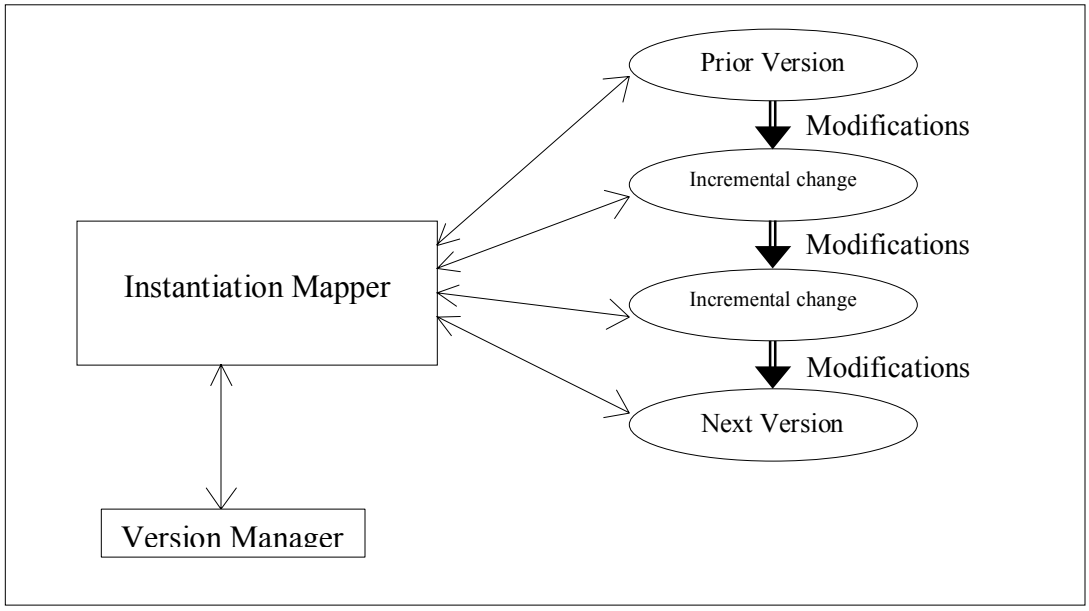


Figure 10.1: Grain Mapping

In order to ensure that the optimizer does not blur the boundary of statements, the programmer can use a special application of the mapping algorithm to couple a set of statements and identify the collection as a special grain called the IDE grain. It should be noted, though, that while the programmer does not need the instantiation mapper for developing the first version, he or she requires a grain mapper for developing versions that have the corresponding prior versions.

10.1.3 Editor

By using an editor, the programmer modifies the duplicated new instance supplied by the instantiation mapper. Then, with the assistance of a special module called the *granulizer*, the programmer can modify the source code as well as manipulate the grain boundaries and grain mapping, as noted in Section 10.1.4.

The programmer can also use this editor to confirm optimization requirements for specially designated IDE grains. If optimization is turned on, the IDE grain boundaries are blurred; if it remains off, IDE boundaries can be identified. The compiler is not going to optimize the IDE grains. Thus, the grain boundaries of IDE grains will remain intact after the compilation. Statements and blocks that are not part of the IDE grains are optimized irrespective of the optimization flag.

10.1.4 Granulizer

By using the *granulizer* interface during the development phase, the programmer can verify grain mappings and grain boundaries. This module must achieve two objectives:

First, in a process called *grain fragmentation*, it divides the source code of the first version into grains. It presents the grain fragmentation information to the programmer through the editor. Whenever the programmer modifies any grain, the granulizer immediately informs the instantiation mapper about the change. The instantiation mapper then stores the latest mapping information. In general, the granulizer recommends that the programmer consider the whole encapsulating function as a single grain. If the programmer manipulates a version other than the first one, the granulizer works differently (See Section 10.2.2). Initially, the instantiation mapper provides the granulizer with grain information for the new instance.

The second objective of the granulizer is to help the programmer in manipulating the grain mappings and the IDE grains. For example, if the prior version of a function has five statements, as shown in Figure 10.2(a), and if the programmer now inserts two statements after the first three statements, as shown in Figure 10.2.(b), the granulizer has the following options to consider:

1. the whole function as a single grain (Figure 10.2(c));
2. the two new statements as a grain (Figure 10.2(d));
3. the third statement and the two new statements as a grain (Figure 10.2(e)); and
4. the two statements as two different statement grains. (Figure 10.2(f)).

The granulizer presents mapping options to the programmer, who chooses the best option. Then it advises the instantiation mapper about the new fragmentation information. In this way, this mechanism ensures that the instantiation mapper always has the latest fragmentation and grain mapping information.

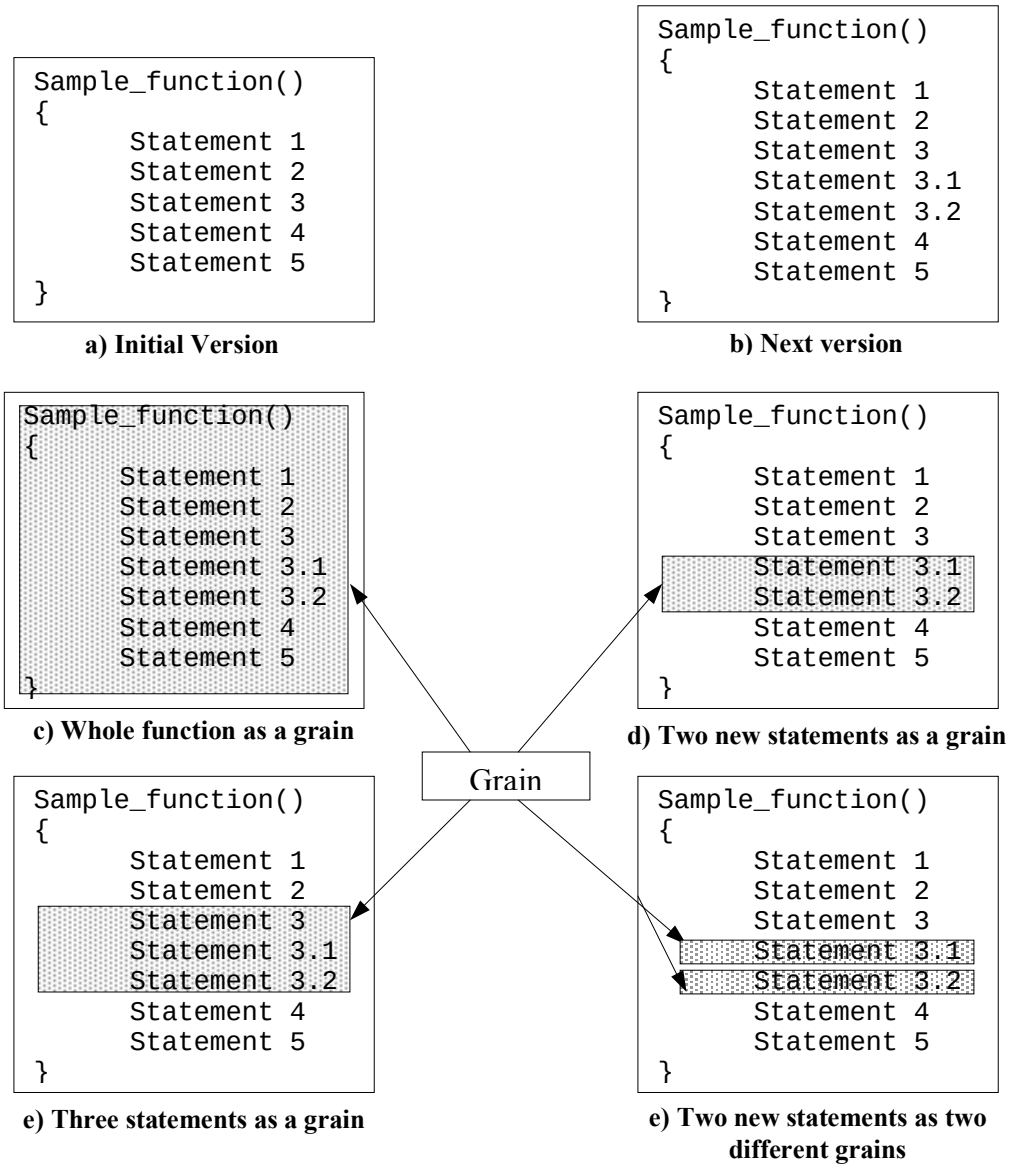


Figure 10.2: Grain Fragmentation Example

10.1.5 Compiler

The compiler translates the program's source code into the corresponding object code. Lexical analysis and syntactic parsing can generate information for grain mapping and fragmentation. A hot swapping enabling compiler requires two distinctive features.

First, this compiler must be able to identify the identifiers whose values the modified grains can affect during compilation. Once this is done, the compiler must generate data regarding their lifetimes as *syntactic information*. Lexical and syntactic analysis can generate such information, which the compiler then passes along to a special module called the *dynamism verifier*. This verifier helps the programmer detect potential incompatibilities and generate dictums using the integral semantic algorithm, which Section 10.1.8 reports on in more detail.

Secondly, the code optimizer in the hot swapping compiler must be able to optimize the program in such a way that the boundaries of all IDE grains still can be located once the process is complete. For this to occur, the granulizer must pass the IDE grain information to the optimizer. Once compilation is complete, the hot spotter assumes control.

10.1.6 Hot Spotter

The hot spotter takes in fragmentation and mapping information from the instantiation mapper and then identifies the differences in source and object codes between the new and old grains. The hot spotter categorizes the grains into three different categories:

- Category 1: new grains that have been created.
- Category 2: old grains that have been deleted.
- Category 3: modified grains from the previous version.

Once categorization is complete, the hot spotter assigns each grain a unique arbitrary identifier necessary for the generation of dictums, and a hot list is created.

Using the hot spotter interface and the granulizer, the programmer can modify the hot list and all mappings. Then the hot spotter passes on all information regarding such updates to the instantiation mapper. If necessary, the program can then be compiled again. Once all tasks are complete and the hot list is finalized, related information is passed to the version manager, and the dynamism verifier assumes control.

10.1.7 Hot Spotter Interface

The hot spotter interface is responsible for presenting the hot list to the programmer, as Figure 10.3 shows. This interface has the following three important tasks:

1. Present the hot list to the programmer in a user-friendly manner.

2. Present grain mapping information using a visual representation that is completely implementation-specific.

3. Allow the programmer to modify the mappings, the hot list, and arbitrary names given to the grains.

The programmer can also provide optimization information to the hot spotter.

10.1.8 Dynamism Verifier

Using the dynamism verifier, the programmer creates the dictatorial. The process consists of three steps:

1. Identification of the integral sections for every leftist present within the hot list;
2. Development of the semantic tree based on the data and control flow graphs; and
3. Generation of dictums with the help of the integral semantic algorithm.

Dictums the dynamism verifier generates help avoid type incompatibilities. Likewise, the integral semantic algorithm can identify potential internal semantic and type incompatibilities. Ultimately, the programmer is the only one who can identify the potential external semantic incompatibilities.

Once the dictums and the dictatorial are generated, the dynamism verifier presents them to the programmer, and the process of generating a new dictatorial begins. If the programmer inserts triggers within the dictatorial, the dynamism verifier passes them to the compiler for syntactic verification. The programmer can also edit the dictatorial and add his own dictums. If, however, they contain compilation errors, the interface returns them to the programmer. The dynamism verifier thus assures the validity of the dictatorial. Only after such validity is assured will the hot packer component be activated.

10.1.9 Dynamism Interface

The distinctive property of this interface is that it presents the dictums and the software architecture to the programmer through a user-friendly visual representation. As soon as this visual representation helps the programmer understand structure, he or she can examine and evaluate the dictums more easily. The interface must also provide the programmer with tools for identifying the integral section's old target grains.

10.1.10 Hot Packer

Using the compiled code and the dictatorial, the hot packer generates the hot pack. First, it compares the prior and next versions of the program and then identifies modified code. Once this is done, it filters machine instructions obtained from the compiler to ensure that only modified code creates the hot pack. Unmodified code is discarded.

10.2 Internal Working

Figure 10.3 illustrates the IDE and the interaction of the ten components that create the hot pack.

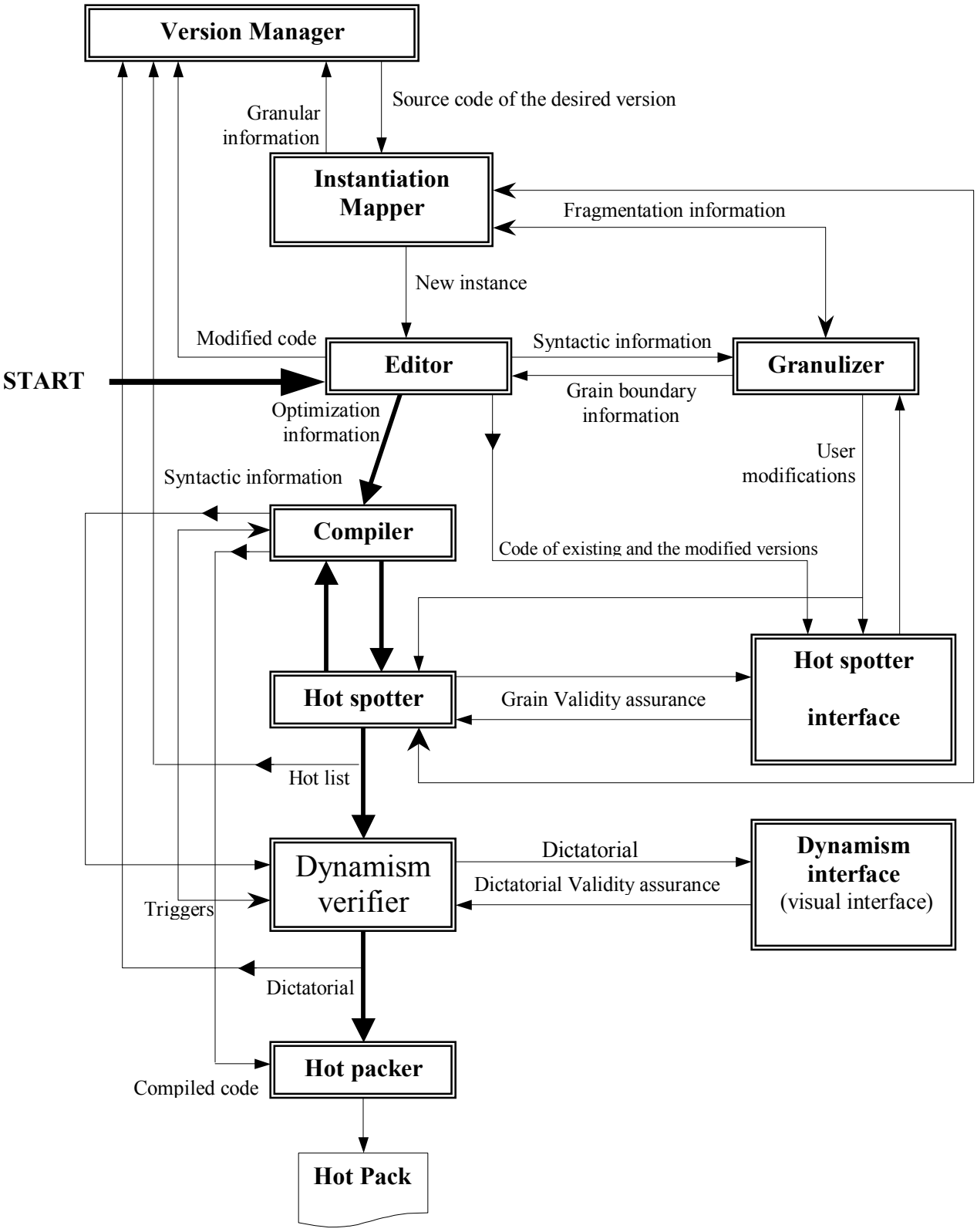


Figure 10.3: Integrated Development Environment

10.2.1 First Version

The development environment has two modes of operation: when the first version is developed (Section 10.2.1) and when the next is developed from the prior one (Section 10.2.2.).

The programmer uses the editor to develop the first version of the program. At this point, the following steps occur sequentially:

1. The granulizer identifies all grains present within the source code and then performs grain fragmentation.

2. As soon as the programmer modifies a grain, the granulizer passes on this fragmentation information. This information can also be delayed until the first version is ready for compilation, but this design implements an instantaneous mapping update method.

3. The granulizer presents the optimization choice to the programmer, who must turn the process on or off. The granulizer passes information regarding the required optimization to the compiler and the instantiation mapper. It should be noted that this optimizing information applies only to IDE grains.

4. The compiler compiles the program and optimizes the object code based on optimization information it receives.

5. Once compilation is complete, because this is the first version of the program, the hot spotter and the dynamism verifier are bypassed, and the hot packer is directly activated.

6. The hot packer creates the executable file based on the compiled code generated by the compiler. This file can be shipped to the client.

7. The instantiation mapper sends fragmentation and optimization information for the first version to the version manager, and the editor passes source code to the same. The timing of this is implementation-specific.

10.2.2 Next version

When the programmer intends to perform hot swapping on a prior version to generate a next version, the following steps take place:

1. The instantiation mapper reads the source code and fragmentation information about the prior version from the version manager and then duplicates the source code to create a new instance. It then maps the corresponding grains of both the duplicate instance and the initial version. Once this is done, the new duplicated instance is transferred to the editor.

2. The editor presents the new source code, plus fragmentation information, to the programmer through a user-friendly interface. In fact, the same grain details of the previous version are displayed in the editor when the duplicated instance is displayed. The programmer then modifies the new instance as required.

3. The granulizer helps the programmer modify the grain mappings (See Figure 10.2). As soon the programmer makes a change, the granulizer sends updated information to the instantiation mapper, which ensures that this mapper always contains the latest available fragmentation information for the code being modified. With this help from the granulizer, the programmer can declare IDE grains.

4. When the programmer finishes modifying the instantiation, he or she can use the editor and the granulizer to review the recommended level of optimization. As soon as the programmer finishes this review, the granulizer sends the optimization information to the instantiation mapper.

5. The compiler translates based on the optimization information acquired from the editor. During this compilation, syntactic information is generated and passed to the dynamism verifier when necessary. When compilation is complete, the hot spotter assumes control.

6. The hot spotter receives grain mapping and IDE grain information from the instantiation mapper and then uses this data to create the hot list.

7. The hot spotter then interacts with the programmer so that he or she can verify and modify grain mappings. If further modifications are required, the hot spotter re-activates the compiler based on the programmer's desire with the new associated information. Step 5 (compilation) is called again with new optimization information.

8. When the hot list is finalized, the hot spotter sends it to the version manager and passes along new mappings to the instantiation mapper. Once this is complete, the hot spotter activates the dynamism verifier and hands it the hot list.

9. The dynamism verifier receives syntactic information from the compiler and the hot list from the hot spotter. Using this information, it creates a semantic graph. The dynamism verifier then comes up with a dictatorial based on the output of the integral semantic algorithm, which resides within it. The programmer uses the dynamism interface to ascertain the validity of the proposed dictatorial.

10. The programmer can use the dynamism interface to add various properties to the transitions and new dictums and triggers to the dictatorial, as well as to ascertain integral sections for the target grains. If there are any errors during the

compilation phase, the dynamism interface prompts the programmer, who must then modify the triggers and ensure the validity of the dictatorial.

11. Once the programmer approves the dictatorial, it is verified and the hot packer is called. The hot packer reads the dictatorial from the dynamism verifier, receives compiled code from the compiler, and combines both to generate the hot pack, which is then shipped to the client's site to perform hot swapping.

12. The version manager receives fragmentation and optimization information from the instantiation mapper, the next version of the source code from the editor, the hot list from the hot spotter, and the dictatorial from the dynamism verifier. The timing of all this is implementation specific.

10.3 Hot Pack Construction

When generating the hot pack, the hot packer has two options. It can package the hot pack with the object code of the whole program and the dictatorial, or it can create the hot pack with only the modified object code and the dictatorial. In either case, the hot pack contains five distinctive sections, as shown in Figure 10.4.

The first section is the *identification section*, an index responsible for specifying hot pack structure. This section contains information regarding the sizes and the locations of all the other sections. The second section is the *identifier table*, which contains information about the various identifiers for the dictatorial and for the trigger functions. Every identifier is associated with an address location corresponding to the executable code within the hot pack. The grain identifiers locate their corresponding grain code present within the grain code section. Similarly every trigger identifier is associated with the corresponding executable code within the trigger code section.

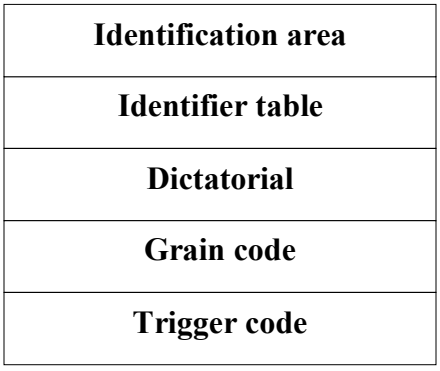


Figure 10.4: Hot Pack Structure

The third section contains the *dictatorial*. The identifiers used within the dictatorial are defined within the identifier table section. The fourth section, or *grain code*, contains all the executable code associated with the grains to be modified. Finally, the fifth section, or *trigger code*, contains code for all triggers defined within the dictatorial.

With the explanation of the mechanism for generating the hot pack now complete, Chapter 11 concentrates on describing how the hot pack is used to perform dynamic modification.

Chapter 11: Hot Swapping Mechanism

11. Introduction

This chapter provides a description of the pseudo-concurrent dual functionality model that must be present at the client's site in order to enable it to read the hot pack's dictatorial and modify the old grains. Section 11.1 concentrates on describing the functioning of the hot swapper within the client environment, while Section 11.2 reports on implementation issues. Finally, section 11.3 provides the detailed information that the hot swapper needs to function as designed.

11.1 Client Environment

As Figure 11.1 shows, the client environment consists of two components, the target process and the hot swapper. The hot swapper reads the hot pack and then modifies the target executing process accordingly. This section describes the mechanism for these two components.

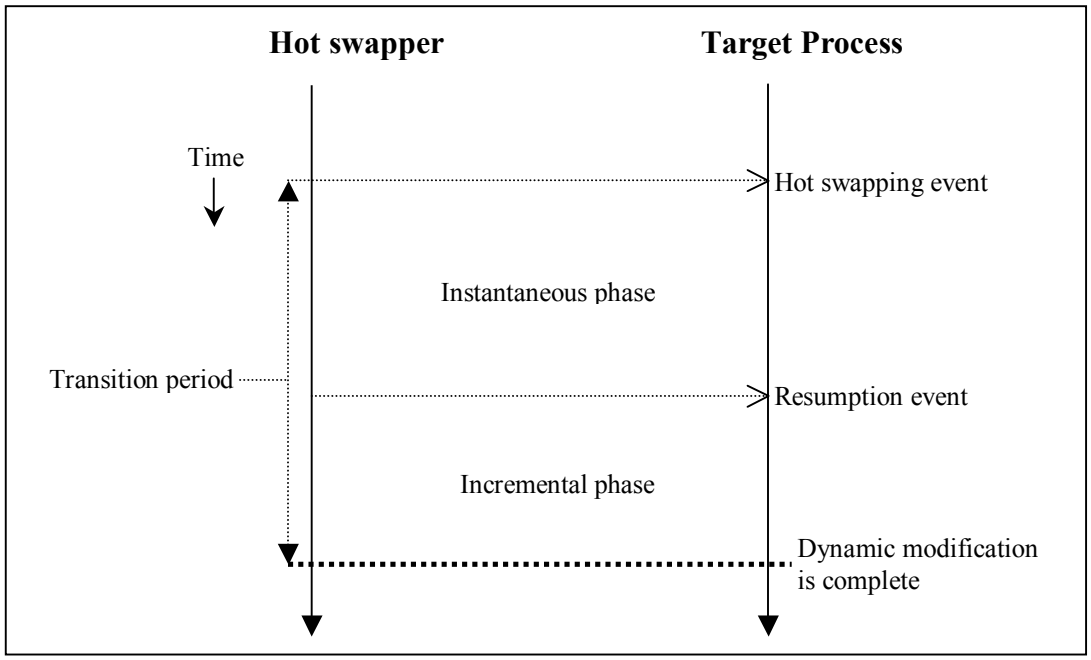


Figure 11.1: Software Hot Swapping Environment

After reading the hot pack, the hot swapper suspends the executing process and then determines whether any of the dictums are satisfied. If any are, the hot swapper modifies the functionality of grain within the suspended target system.

When all executable dictums are processed, the suspended target system resumes execution. If the hot swapper identifies certain dictums that cannot be executed during this, the *instantaneous phase*, it delays the modification process for those dictums, and the target system resumes, with the hot swapper monitoring the location of the execution

pointer. As soon as the pointer exits the integral section of the associated grains, the hot swapper then modifies the grains associated with the leftover dictums during what is called the *incremental phase*.

The hot swapper has three components: the *dictatorial analyzer*, the *instantaneous component*, and the *incremental component*, as shown in Figure. 11.2.

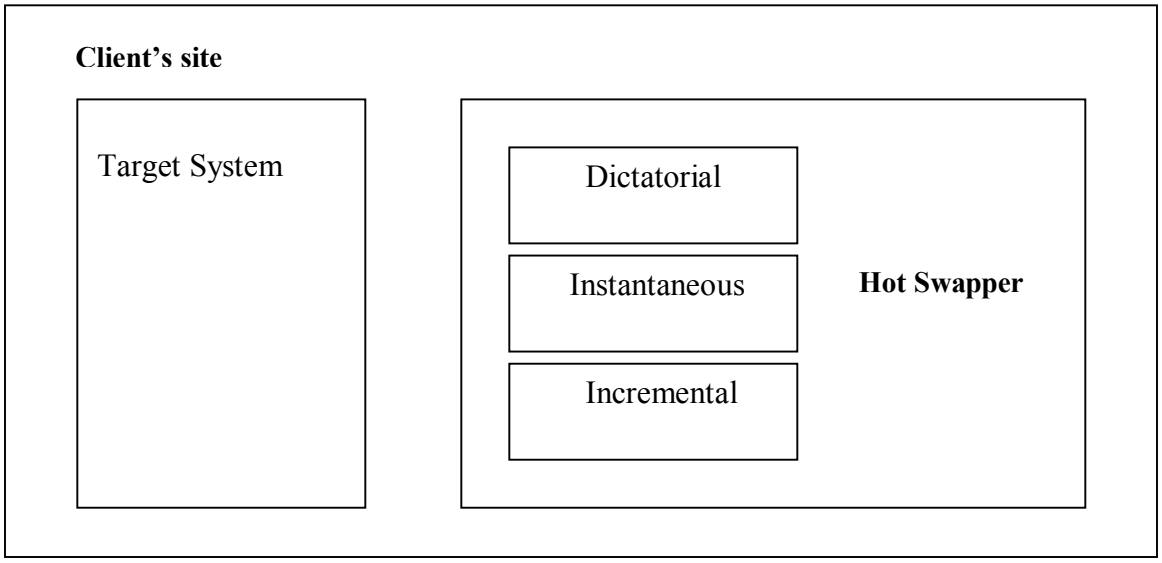


Figure 11.2: Software Components Within the Hot Swapping Environment

The dictatorial analyzer is the first component activated when the hot swapper is invoked. It reads the hot pack and then assumes control of the hot swapper's other two components. As soon as the dictatorial analyzer reads the hot pack, it creates a chronological order of events to be performed during hot swapping. This series of events is based on the order in which the dictatorial of the hot pack determines the dictums must be executed.

After constructing the chronological order of events, the dictatorial analyzer invokes the instantaneous component, which suspends the executing target functionality. As soon as the hot swapping event occurs, the whole system enters the instantaneous phase. The instantaneous component examines the address space of the suspended target system and gathers the *target information*. Target information contains locations of the functions within the text area and the information about the garbage. Section 11.3 provides a detailed description of target information.

During the instantaneous phase, the related component generates and then sends the target information to the dictatorial analyzer, which verifies the information. After this verification, the dictatorial analyzer categorizes the target grains as either *modifiable* or *unmodifiable*. In general, modifiable target grains constitute the instantaneously unused grains discussed in Chapter 3, Section 3.3. The dictatorial analyzer then develops a strategy based on the dictatorial and the grain categorization information to modify the

categorized target grains. The modifiable grains are modified immediately, but the integral, suspended and executing grains (unmodifiable grains) cannot be modified until the execution pointer reaches the corresponding safe regions, as indicated by the associated dictums. The target information is updated whenever a modification is made.

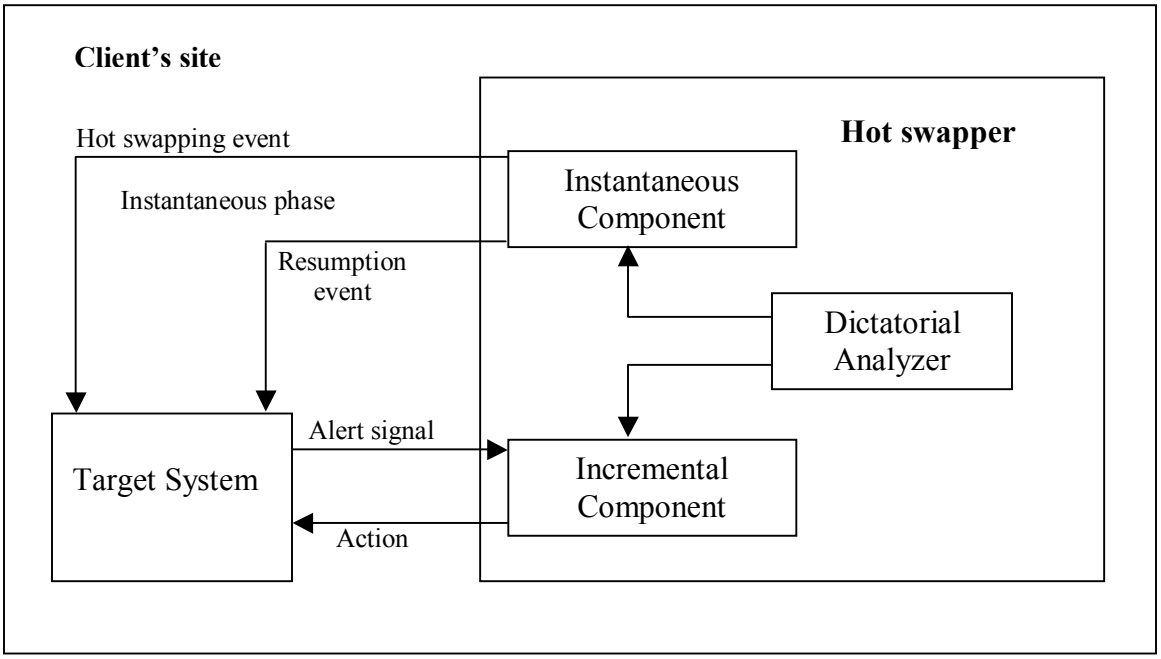


Figure 11.3: Process of Software Hot Swapping

The dictatorial analyzer informs the instantaneous component about modifiable and unmodifiable grains. The instantaneous component then performs the following four activities:

1. It modifies the modifiable grains.
2. It updates the target information.
3. It initiates the execution monitoring mechanism, as described below.
4. It permits the suspended target system to resume execution.

Once these tasks are complete, control passes to the dictatorial analyzer, which initiates the incremental component for monitoring execution of the target system.

The *execution monitoring mechanism* helps the incremental component monitor the location of the execution pointer during the incremental phase. The underlying principle of this monitoring mechanism is to alert the incremental component whenever the execution pointer reaches safe spots important to the integral sections of the suspended and executing grains. These safe spots can be located at the ends of integral sections, at the ends of functions, or at points the programmer designates.

Once the execution pointer reaches a safe spot, the incremental component suspends execution of the target functionality and then alerts the dynamism verifier about the pointer's new location. This verifier checks to see if any of the dictums can be satisfied. If they can, it instructs the incremental component accordingly. The incremental component then modifies the indicated target grains within the address space of the executing functionality. It also updates target information during this process. The target functionality then resumes execution. This system of monitoring continues until all dictums are satisfied and their corresponding grains are updated.

There can be situations in which the dynamism verifier determines that none of the grains can be modified. In this case, the incremental component simply allows the process to resume execution. Once hot swapping is complete, the incremental component will terminate the monitoring mechanism. Clearly, the monitoring mechanism is highly dependent on the implementation and the process models. Section 11.2 below provides a description of the implementation issues.

However, it is important to note that if the instantaneous component modifies all target grains during the instantaneous phase, the incremental phase will not be required. It exists only if the dictatorial analyzer identifies suspended or executing grains during the instantaneous phase.

11.2 Implementation Issues

This section provides a description of implementation issues involved in the design of the client environment. Section 11.2.1 reports on the process models that can be used to implement the client environment. Section 11.2.2 reports on the monitoring mechanisms, and Section 11.2.3 concentrates on describing issues behind the implementation of both mechanisms.

11.2.1 Process Models

The hot swapper and the target system can be implemented in three different ways: the *brute-force* method, whereby the functionalities are designed as two different autonomous processes; the *single process* method, also known as the integrated model; and the *co-routine model*, in which the two functionalities are implemented as separate co-routines.

11.2.1.1 Brute-force Model

In the brute-force model, the target and hot swapper functionalities are implemented as two autonomous processes within the operating system. Whenever hot swapping is required, the hot swapper starts execution, then suspends the target process, performs the instantaneous phase modification, and, if necessary, induces the execution monitoring mechanism. Finally, the target system resumes execution. If the execution monitoring mechanism is initiated, the incremental component of the hot swapper monitors the execution of the target process until the incremental phase is complete.

This model is much simpler than the other two process models. To perform software hot swapping, the hot swapper must use some of the operating system dependent application program interfaces (APIs). The disadvantage of this method is that operating system calls are generally inefficient; each must undergo a number of security checks before finally executing. This kind of a limitation makes the process model operating system specific. Also, this model works on operating systems that support calls allowing a process to suspend a different process. The operating system APIs required for implementing the brute-force method are

- APIs for suspending and resuming the execution of a process.
- APIs for reading the address space of the suspended process.
- APIs for writing into the address space of the suspended process.
- APIs that allow address space resizing.

Most of the POSIX based operating systems, including Linux, Unix, and Solaris support these system calls; hence, they support the brute-force model. As further explained in Chapter 12, the author has chosen the brute-force model for the prototype program to demonstrate software hot swapping, because the implementation of the brute-force model t is simpler than the implementation of the other two models.

11.2.1.2 Integrated Model

In the integrated model, the target functionality and the hot swapper execute as a single process. The target functionality executes within the address space of the hot swapper, which acts as a wrapper to encapsulate the executing functionality, as shown in Figure 11.4.

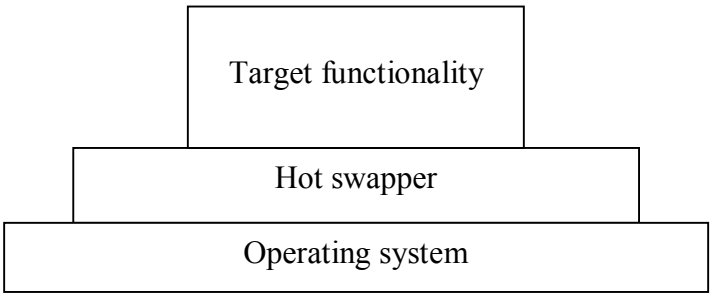


Figure 11.4: Integrated Model

Whenever hot swapping is required, the hot swapper reads the hot pack and performs as required on the target functionality. When the vendor compiles the program, all necessary code for the hot swapper is stored within the executable file so that during execution the hot swapper encapsulates the target functionality. The advantage of this method is that in it, dynamic modification occurs independently of the underlying operating system, which supports an API that allows processes to modify their page protection. Whenever a process is loaded into memory, the pages storing the text area of the process are write-protected so that if an attempt is made to perform a write operation over them, the system crashes. Hence, the operating system must provide a system call to permit a process to

modify its own page protection, thus also allowing the hot swapper to modify the text area of the target functionality.

11.2.1.3 Modified Co-routine Model

When a program in which this model has been embedded is loaded, the wrapper functionality behaves like an execution moderator and begins execution. The wrapper can be thought of as the controller of the entire address space allocated to that program. The target system starts as a *prong* on top of this wrapper, a functionality that has executable code and a state associated with that code. The execution of prongs over an address space is similar to the execution of processes over the operating system.

In this model, every process has two permanent prongs: the target prong, containing the target functionality, and the hot swapper. The wrapper provides execution time for the prongs based on a deterministic priority that the programmer can modify.

Whenever a program with this model embedded in it begins execution, the wrapper functionality behaves like a process scheduler. During normal execution, the target system prong is given the highest priority. During the initial development, however, the programmer can assign dynamic priority to both these prongs.

Whenever software hot swapping is initiated, the wrapper allows the hot swapper prong to execute. The hot swapper reads the hot pack and then modifies the target prong accordingly. This execution model is similar to a small computational world within the executing process. Inside the process, there are different functionalities that execute based on specific need, as Figure 11.5 shows.

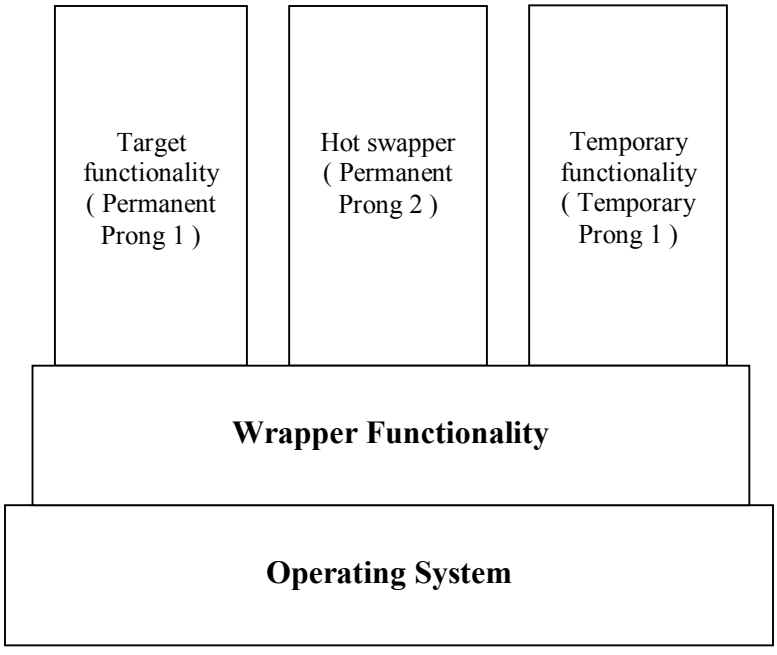


Figure 11.5: Modified Co-routine Model

One advantage of this model is that no operating system security hinders software hot swapping. The only security issue involves page protection. It is the responsibility of the wrapper to execute efficiently, but it is reasonable to argue that the efficiency is higher, because the operating system is not a part of this system. Another advantage of this model is its maximum flexibility for programming the execution monitoring mechanism and configuring the hot swapping methodology. All the hot swapping APIs necessary for the programmed communication path can be embedded within the wrapper functionality.

The modified co-routine model is an ideal model that can be used to hot swap an operating system, because the entire system resides on top of the wrapper functionality as a permanent prong. A separate hot swapper prong can modify the executing operating system on the fly. However, implementing this kind of model is tricky. This author, therefore, argues that since most of the library routines (thread libraries) that produce it are operating system-dependent, the modified co-routine model is realistic only if the wrapper is programmed using assembly language. The model's internal working is completely platform independent.

11.2.2 Execution Monitoring Mechanism

The execution monitoring mechanism is crucial to software hot swapping, because the incremental component must use it to locate the execution pointer within the address space. Moreover, the efficiency of the target functionality's execution during the transition phase depends on this mechanism.

Despite its value, though, some execution overhead is associated with the monitoring mechanism. This overhead must be reduced in such a way that:

1. Normal execution efficiency must not be greatly affected, and
2. Execution efficiency of the target during transition remains acceptable.

Sections 11.2.2.1 and 11.2.2.2 provide a description of the two different ways of implementing the execution monitoring mechanism.

11.2.2.1 Controlled Execution

Using a controlled execution method, the incremental component controls execution of the target system. The target system is permitted to execute until the execution pointer reaches a safe spot. The incremental component then alerts the dictatorial verifier, which instructs the incremental component on which action to take. For example, a `ptrace` system call can be used in POSIX-compatible operating systems so that a single process can actually control others. This controlled execution method can also be implemented within other models; however, in order to make the pseudo concurrent dual functionality model feasible, the brute-force method should be used.

11.2.2.2 Crumbs

Crumbs are small pieces of code that can be placed at different memory spots to trap events when the movement of the execution pointer causes an active (or integral) grain to become an IUG or an IUG to become an active or integral grain. Crumbs, which can be inserted during the compilation process or the instantaneous phase, have two states: *active* and *inactive*. Although, crumbs and break points are similar in concept, the hot swapper can activate and deactivate crumbs.

In general, a crumb contains two parts: a header and a body, as shown in Figure 11.6.

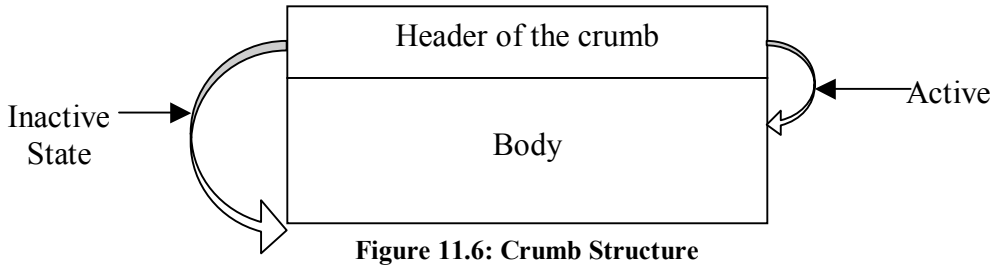


Figure 11.6: Crumb Structure

When the compiler inserts a crumb during the compilation process, it is translated into an inactive state, to be initiated during the instantaneous phase. Here, the purpose of the crumb is to trap at implicit safe spots certain crucial events, such as events when the execution pointer comes out of an integral section or enters an integral section or when an active or integral grain turns into IUG and so forth.

Whenever the execution pointer enters the scope of a crumb, the crumb informs the incremental component, which in turn alerts the dictatorial verifier so that it can instruct the incremental component accordingly. When the incremental phase is complete, the incremental component deactivates all active crumbs, generally by setting the crumb header in such a way that it inserts a `jump` instruction to make the execution pointer bypass the body. The instantaneous component reactivates such crumbs the next time the hot swapper attempts hot swapping.

In order for the instantaneous component to locate crumbs generated during compilation, a special table containing their addresses is generated at the same time. If, however, the instantaneous component inserts the crumbs into the address space, then the instantaneous component must pass the location information to the incremental component so that it can remove them once the modification is complete. Once transition is complete, this instruction can be set to a `jump` instruction once again so that the crumb becomes inactive. Of course, the time at which the crumb can be deactivated is implementation specific.

11.2.3 Implementation Feasibility Issues

The client environment must contain the target system and the hot swapper, which are implemented using one of the three process models described in Section 11.2.1. In addition, every process model needs to have an execution monitoring mechanism. Whenever the two mechanisms described in Section 11.2 are integrated with the process models, many implementation issues arise.

11.2.3.1 Brute-Force Method Implementation

In this model, because the incremental component uses the operating system features to perform the incremental phase, it is largely inefficient. In order to regain efficiency, it is necessary to minimize usage of the operating system features.

The instantaneous component suspends the target system, modifies all possible grains, and then implements the incremental component depending upon the choice of execution monitoring mechanism. Irrespective of the mechanism chosen, the overhead incurred by the instantaneous phase is straightforward in the sense that it is deterministic.

The overhead the incremental phase incurs depends upon the configuration when the hot swapper attempts hot swapping. The duration of the incremental phase is non-deterministic. If the controlled execution mechanism is used, all the associated execution controlling functionality must be incorporated within the incremental component. The incremental component allows the target functionality to execute until the suspended or executing grains become IUGs. It then alerts the dictatorial analyzer and acts accordingly. In order to control the execution, the incremental component uses the operating system features, such as the `ptrace` system call. This has an associated overhead.

If crumbs are used for the monitoring mechanism, then the instantaneous component must activate them and allow the suspended target system to resume execution. Depending upon the features the operating system provides, activated crumbs alert the incremental component through either inter-process communication or signals. Both methods are inefficient.

There is, however, a second method of implementing crumbs that involves inserting the incremental component into the target process during compilation. Whenever the hot swapper attempts hot swapping, the instantaneous component modifies all possible grains and then inserts the chronological order present within the dictatorial analyzer into the incremental component. The instantaneous functionality activates all crumbs and lets the executing process resume execution. Whenever the execution pointer encounters a crumb, it then jumps to the incremental component present within the same address space. Based on the chronological order initially generated by the dictatorial analyzer, the incremental component performs modifications. Once the transition is complete, the incremental component deactivates the associated crumbs. The target information of the grains is thus modified in conjunction with the incremental phase. This author has implemented this model, which Chapter 12 reports on in detail, in his prototype program.

11.2.3.2 Integrated Model Implementation

As explained in Section 11.2.1.2, the integrated model encapsulates the target functionality within the address space of the hot swapper. Since both the executing functionality and the hot swapper are present within the same address space, the system is implemented independently of the operating system. The only possible problem that can arise is that of text area write-protection.

Controlled execution can be used within a system based on the integrated process model. Whenever hot swapping is required, execution control is passed to the hot swapper. Both the hot swapper and the dictatorial analyzer are activated simultaneously, and, after generating the chronological order of events, the instantaneous component is likewise activated. Once all possible grains are modified, the incremental component controls execution of the target functionality in order to perform the incremental phase.

Crumbs can be used to simplify implementation of the incremental component. Since the incremental component is within the same address space as the target, the associated overhead involved in the monitoring the target is less than that of the brute-force method. However, with the integrated model, the execution characteristics of crumbs and controlled execution are not precise.

11.2.3.3 Modified Co-routine Model Implementation

The *extended co-routine system* can be implemented by modifying the existing co-routine process model. Here, the programmer can design temporary prongs for his or her own purposes. Since the executing functionality and the hot swapper functionality are present within the same address space, the system can be implemented independently of the operating system. Again, a potential problem involves text area write protection.

This model can be used to hot swap an operating system. The system can be implemented as the target functionality and the hot swapper can be implemented as another permanent prong. Further examination of this model, however, is left for future work.

11.3 Target Information

During hot swapping, maintaining address space information is vital. The contents and number of tables present within the address space information can depend on the design, but a general hot swapping technique requires such data in order to perform dynamic modification. Without the information, the dynamism modifier simply cannot categorize modifiable and unmodifiable grains during the instantaneous phase.

Section 11.3.1 provides a description of the first table, the *garbage collection table*, which contains all available islands of unused memory that result from previous dynamic modifications. Section 11.3.2 reports on the second table, which locates all grains that are potentially hot swappable.

11.3.1 Garbage Collection Table

The first table, the *garbage collection table*, lists all islands of unused memory within the address space of the process. Whenever the hot swapper removes grains, it records the freed memory space within this garbage table. Typically, as Figure 11.7 shows, this garbage table contains two columns, *begin* and *end*.

Begin	End

Figure 11.7: Garbage Table

Each row contains the starting and ending addresses of every island of unused memory. As soon as a transaction occurs, the hot swapper updates this table. Whenever a grain must be modified, the hot swapper checks the garbage table to determine whether free space exists for the rightist of the transition. If it does, the hot swapper copies that grain into the available location.

11.3.2 Grain Location Table

The grain location table contains information regarding the location of grains in the target functionality. As in Figure 11.8 shows, this table typically contains four columns. The first column contains the grain identification, an arbitrary key uniquely assigned for each grain. The second and third columns contain the starting and ending addresses of the corresponding grain, respectively. The fourth column contains information about the grain's activation state. Function preambles and postambles that the compiler inserts during the compilation phase perform the process of incrementing and decrementing the byte (activation byte) stored within the activation state column. The activation byte is increased by one (incremented) whenever the execution pointer enters the scope of the function. Similarly, whenever the execution pointer exits the scope of the associated function, the activation state byte is decreased by one (decremented). The hot swapper needs to know the grain location information and the activation state before performing hot swapping.

Grain ID	Begin	End	Activation State

Figure 11.8: Grain Location Table

11.4 Crumb Placement Strategy

During the incremental phase, the placement and number of active crumbs within the address space can affect the execution efficiency of the target functionality. This section describes the arguments related to the crumb placement strategy.

As explained in Chapter 5, the hot swapper can modify every target grain only when the execution pointer is outside its associated integral section, which is comprised of a set of contiguous code regions, each called an *integral region*.

During the instantaneous phase, the dictatorial verifier categorizes a target grain as unmodifiable if the execution pointer resides within an integral region of the respective integral section. Hence, the instantaneous component must delay modification of such a grain until the execution pointer exits the scope of the associated integral region, which can occur only when the suspended functionality resumes execution. Since the incremental component must be able to suspend the execution of the target functionality just as the execution pointer exits the integral region, the instantaneous component must place the crumb at the end of the integral region within whose scope the execution pointer is currently present.

Chapter 12: Research Prototype

12. Introduction

Chapter 12 provides a discussion of the prototype program that this author has created to demonstrate a limited form of software hot swapping. Developed as a result of this research, this program proves that software hot swapping is a viable concept and demonstrates the ease and feasibility of implementing the technique. This chapter also makes the argument that this technique can be extended to a full-fledged development environment.

For the purpose of this prototype demonstration, this author has used a special imperative language — SynC+ — , which is semantically similar to the C language and which Section 12.1 reports on in detail. Additionally, the prototype contains two important components: the integrated development environment or *IDE*, which this author also designed to compile SynC+ programs, described in Section 12.3 and the hot swapper, described in Section 12.4. Finally, Section 12.5 reports on some of the test cases used to demonstrate software hot swapping.

12.1 SynC+

In fall 1999, Xiaoling Bao, Zhanming Qin, Mohammed Benhsain, Baoping Zhang, Rohit Gupta, Emre Tunar, Pradeep Tumati, and Deepak Gupta, all students at Virginia Tech, initiated the design of SynC+, an imperative language, as part of a compiler course project (CS 5304). By first demonstrating that software hot swapping can work with SynC+, this author expects to demonstrate that it is possible to extend the technique to a language like C.

A SynC+ program is composed of a single source file and can contain any number of functions. Since the program's entry point is through the main function (`main()`), every program must contain a main function. Header files and external functions are not supported in SynC+. However, a few external functions have been hard-coded within the grammar and are linked to the C library during the link-load period. The programmer can use these functions to develop the program.

Character set

A SynC+ file is a sequence of characters selected from a set. A SynC+ compiler may use any character set as long as it includes at least the following characters:

- 26 lower case alphabetical characters: a b c d e f g h i j k l m n o p q r s t u v w x y z;
- The ten decimal digits: 0 1 2 3 4 5 6 7 8 9;
- The blank or space character; and

- Fifteen graphical symbols:

! exclamation point	. period	* asterisk
(left parenthesis	) right parenthesis	– hypen or minus
{ left brace	} right brace	; semi colon
= equal	+ plus	/ slash
> greater than	< less than	, comma

A SynC+ program has to be divided into lines which the compiler restricts in size to 80 characters.

Operators and separators

SynC+ has a simple set of five main arithmetic operators:

+ Addition,
- Subtraction,
/ Division,
* Multiplication, and
= Assignment operator.

In addition, the language possesses six conditional operators:

>> Greater than,
<< Less than,
>= Greater than or equal to,
<= Less than or equal to,
== Equal to, and
!= Not Equal.

Identifiers

In this language, an identifier is a sequence of alphabetic characters or digits with a maximum length of 8. The identifier's first character must be a member of the alphabet.

Identifiers of variables

Every identifier can be associated with a type integer or a character type. An integer variable is a 16-bit word, while a character variable is allocated ten bytes. Hence, it is possible to store a string of 10 characters identified by a name. `int` is used to declare a variable of the integer type. `char` is used to declare a variable of the character type.

Functions

Functions in SynC+ take in and return a single argument. The parameter-passing mode is the *call by value method*. The function return value and the function parameter types can be of three types:

- `void` if the function does not require an argument or does not return a value;
- `int` integer argument or return value; and
- `char` character argument or return value.

It is mandatory that the main function's argument and the return type be of the type void (`void main(void)`). A body enclosed within curly braces must define every user-defined function.

Apart from the user-defined functions, SynC+ also contains some external functions that the programmer can use to perform input and output:

`cin variable , variable... ;`
Used for input, this function call links with the `scanf` function (C library).

`cout variable , variable…;`
Used for output, this function call links with the `printf` function (C library).

`init();`
This function initializes the SVGA graphic engine [EX 14]. It links with the `vga_init()` function, which is a part of the SVGA graphic engine.

`plot( x coordinate , y coordinate , color );`
The programmer can use this function to display a pixel on the screen. It links with the `vga_drawpixel` function, which is part of the SVGA graphic engine. The color parameter of the `plot` function is used to initiate another function, `vga_setcolor`, which sets the specified color. Hence, whenever the `vga_drawpixel` is called, the graphic driver displays the pixel with the color specified by the `vga_setcolor.`

`line( left x coordinate , left y coordinate ,`
`right x coordinate , right y coordinate,`
`color );`
This function is used to display a line on the screen and links with the `vga_drawline` function, which is part of the SVGA graphic engine. The color part of the `line` function is used to initiate another function, `vga_setcolor`, which sets the specified color. Hence, whenever the `vga_drawline` is called, the graphic driver displays the line with the color specified by the `vga_setcolor()`

`getch();`
`getch` makes the system wait until the user types in a key; it links with the `vga_getch`.

clear();

clear makes the graphic engine clear the screen and links with vga_clear, which is a part of the SVGA graphic engine.

Syntax

The syntax of SynC+ is shown below. The start symbol is the <program>.

```
Start Symbol
      Program ->function_definition | variable_declerations

Functions
      Function_definition  ->  function_type identifier (parameter_list)
                               block
      Function_type -> void | char | int
      Parameter_list -> void | int identifier | char identifier

Blocks and Statements
      Block -> {statement_list | Block}
      Statement_list -> statement statement_lst
      Statement_lst -> statement statement_lst | statement | NULL

      Statement ->  expression;
                  | if_block
                  | while_block
                  | ext_funs
                  | return expression;
                  | variable_declerations;
                  | function_call;

External functions
      Ext_funs ->   cin identifier_list
                  | cout identifier_list
                  | init( );
                  | getch( );
                  | line(identifier, identifier, identifier,
                    identifier, identifier );
                  | plot(identifier, identifier, identifier);
                  | clear( );

Other non-terminals
      if_block -> if ( expression ) block else_part
      else_part -> else block | NULL

      variable_declerations ->  int identifier_list
                              | char identifier_list
      identifier_list -> identifier id_list
      id_list -> , identifier id_list|;
      identifier -> character id    { The size of an identifier is
                                      restricted to 8}
      id -> alphanumeric id| character id | NULL

      while_block -> while (expression) block;
```

```
function_call -> identifier(parameter_list)
```

Expressions

```
Expression -> identifier = item | item | item conditional item
Item -> identifier
      | constants
      | (expression)
      | item symbol item
      | function_call
```

{The semantic phase of the compilation process implements associativity and operator precedence.}

```
symbol -> + | - | *
conditional -> << | >> | <= | >= | == | !=
alphanumeric -> character | number
number -> 1|2|3|4|5|6|7|8|9|0
character -> a|b|c|d|e|f|g|h|I|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
```

12.2 Platform

The computer system used for this prototype program is a Pentium III machine with a processor speed of 600Mhz. The test bed operating system is Linux [EX 16, 17], an open source operating system that can be customized easily. As an operating system, Linux is similar to Unix, as it follows the POSIX1003.1 standard notation and supports true preemptive multi-tasking. Linux allows a number of users to work simultaneously and, from Version 2.0 onward, supports multi-processing. In addition, Linux runs on several architectures, supports demand load executables, and provides all the features a typical Unix OS offers. Each page on x86 architecture is 4KB.

Linux also uses system calls similar to Unix. A system call occurs when a user process requests a service provided by the kernel. At this point, the user process is put on hold while the kernel examines the request, attempts to carry it out, and then passes the result back to the user process, which then resumes execution. System calls generally guard access to resources that the kernel manages. The largest categories of system calls are those that deal with managing I/O, processes, times, memory, and file systems. This section discusses some of the system calls used by Software Hot Swapping.

The most important system call is the `sys_ptrace` system call, which lets one process control another. In this way, by running another process, the user can issue a `sys_ptrace` command and control the target process. Through controlling the process, the user can execute the target process step by step and perform memory reads and writes. The hot swapper works based on this principle. The hot swapper controls the execution of the target process, thereby performing modification. GNU debugger (GDB) also uses this system call, which takes four arguments:

```
int sys_ptrace(LONG request, LONG pid, LONG addr, LONG data);
```

The function processes various requests defined in the first argument, called `request`. The second argument is the process identity of the target process; the third, the address field on which the intended request is to operate; the fourth, the data field used in case the request specifically says that the passed argument has to be written in the address space, as specified by the third argument. The fourth argument consists of optional data. The various requests that can be passed as the first argument are:

PTRACE_TRACEME
Here, a process can specify that its parent process controls it via `ptrace( ).`

PTRACE_ATTACH
By calling this request, the calling process can make any process its child and stop its execution. The main criterion here is that the user_id and the group_id must be same. All the following requests can act only after attaching a process, except **PTRACE_KILL**.

PTRACE_PEEKTEXT
This request lets the process read a 32-bit word from the control process memory area.

PTRACE_POKETEXT
This request lets the process write a 32-bit word into the control process memory area.

PTRACE_PEEKUSR
This request enables a long value to be read from the user structure for the process.

PTRACE_POKEUSR
This request enables a long value to be written on the process registers from the user register structure, in order to modify the process registers.

PTRACE_CONT
After being interrupted by a signal, this request can continue the child process. The argument `data` can be used to decide which signal the process will handle when it resumes execution. Once the child process receives the signal, it informs the parent process and halts. The parent process can now continue the child process and decide whether it should handle the signal. If the data argument is null, the child process will not process any signal.

PTRACE_SINGLESTEP
This differs from **PTRACE_CONT** in setting the processor's trap flag. The process thus executes only one machine code instruction and generates a debug interrupt. This sets the **SIGTRAP** signal, which is then interrupted again. In other words, this request allows the machine code to be processed instruction by instruction.

PTRACE_SYSCALL
This request causes the child process to resume in the same way as **PTRACE_CONT,** but only until the next system call. The process arrives at the next system call, where it halts and receives the **SIGTRAP** signal. At this point, the parent process can inspect the arguments for the system call.

PTRACE_DETACH
This is the converse of **PTRACE_ATTACH**: it detaches the target process and makes it the child of the process of its original parent.

PTRACE_KILL
This terminates a process without having to be attached to it.

12.3 Integrated Development Environment (IDE)

This section provides a description of the structure of the prototype program that accompanies this research. Since the IDE must support the development of both the first and the next versions, it contains an editor, a compiler, an assembler, a version manager, and a hot spotter. The underlying principles behind the IDE differ depending upon the version under consideration. A description of the generation of the first and next versions follows.

In order to develop the first version, the programmer uses the editor to generate the program, completes its development, and initiates the compiler. The compiler translates the program and generates the equivalent assembly language instructions, as well as the required grain-mapping information, which is passed to the version manager. The open source Netwide Assembler (NASM) translates the assembly language instructions into machine instructions and then produces the equivalent executable file [EX 15]. Figure 12.1 shows the prototype architecture for developing the first version.

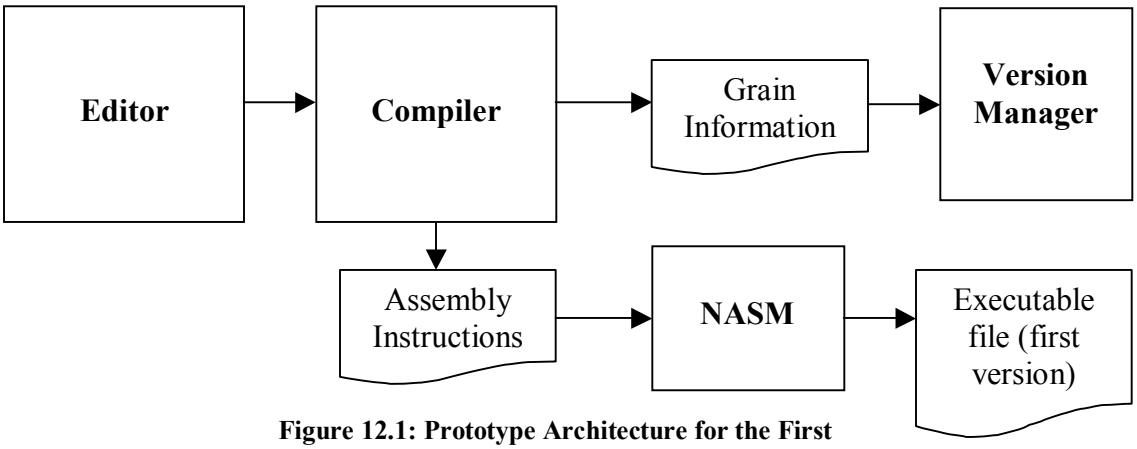


Figure 12.1: Prototype Architecture for the First Version Development

When the programmer wants to develop the next version, he or she issues a command to the version manager, which passes grain information and source code for the first version along to the editor. Since the editor is tightly coupled with the compiler, it notes all modifications the programmer makes to the prior version.

Whenever the programmer wants to generate the hot pack, the editor passes the grain information and the source code to the compiler, which then generates the assembly language program in such a way that the IDE can identify all the instructions that have been modified since the prior version. The NASM then assembles the program and

generates a machine instruction list file that contains the assembly language program instructions, as well as their corresponding machine instructions. The IDE component that activates the assembler fetches this listing file and sends it to the hot spotter, which reads the listing file, identifies all the modified instructions, and then creates a hot pack with all the new machine instructions that correspond to pre-specified memory addresses. This hot pack is then passed to the hot swapper. Figure 12.2 shows the prototype architecture for developing the next version.

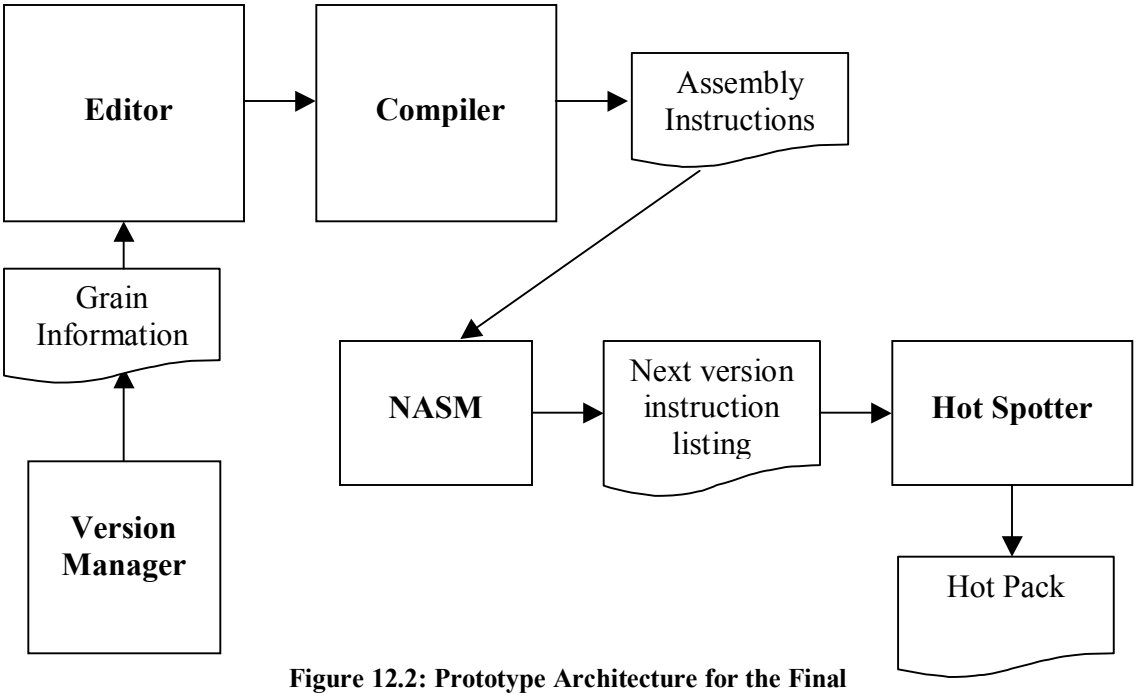


Figure 12.2: Prototype Architecture for the Final Version Development

12.4 Hot Swapper

Since the process model used for the prototype is the *brute-force model*, the author implements the hot swapper as a separate process. When the hot swapper reads the hot pack, it then suspends the target process the author has already initiated by using the `ptrace` system call. The contents of the hot pack are then copied into the text area of the suspended process and execution resumes. It should be noted that the prototype does not support an incremental phase. The hot swapper uses the following algorithm:

Step 1: Read the hot pack.

Step 2: Read the `renenv` file, which contains the process id (pid) of the target process. The target process creates this file whenever it starts execution. The `runenv` file also contains the starting address of the text area of the target process. Section 12.5 provides a more detailed description of the significance of the `runenv` file.

Step 3: Use the `ptrace` system call to suspend the target process.

Step 4: Copy the contents of the hot pack into the text area.

Step 5: Let the suspended process resume execution by using the `ptrace` system call.

12.5 Demonstration

The demonstration of the prototype program occurs in two phases. The programmer develops the first version in the first phase and then modifies the source code of the program the second version. The IDE generates the hot pack, which constitutes the second phase. Finally, the hot swapper reads the hot pack and then upgrades the text area of the executing process. This section provides an explanation of the process.

In order to develop the first version, the programmer must open the IDE, which issuing the following command at the C prompt can initiate:

`./a.out` (This command initiates the IDE)

The programmer needs to enter his or her program into the editor. For the purpose of the demonstration, this author has chosen the program shown in Figure 12.3.

```
// Main function
Void main(void)
{
      int a,b;
      a=1;
      b=2;
      while(1!=2)
      {
            temp();
      }
}

// This function displays numbers from 1 to 5 whenever called
void temp(void)
{
      int a,b;
      a=1;
      b=1;
      while(a<=5)
      {
            cout a;
            a=a+1;
            // The following loop reduces the speed of execution
            while(b<=1000000)
            {
                  b=b+1;
            }
            b=1;
      }
}
```

Figure 12.3: First Version Program

The purpose of this program is to display numbers from one to five repeatedly in a sequence. The `main()` function contains a unending loop. The function `temp()` is called every time the execution pointer iterates within the scope of the unending loop. The `temp()` function contains a scoped accumulation operation, which, when called, counts and displays numbers from one to five. The programmer can then compile this program by pressing the F1 key. At the completion of compilation, the IDE generates the assembly program, whose structure Figure12.4 depicts, and a *make* file. The *make* file contains instructions that execute the NASM, which then generates the executable file from the assembly program. The name of the executable file is `einitial`. After compilation, the programmer can press the ESC key to terminate the IDE and type ./`einitial` at the command prompt to execute `einitial`.

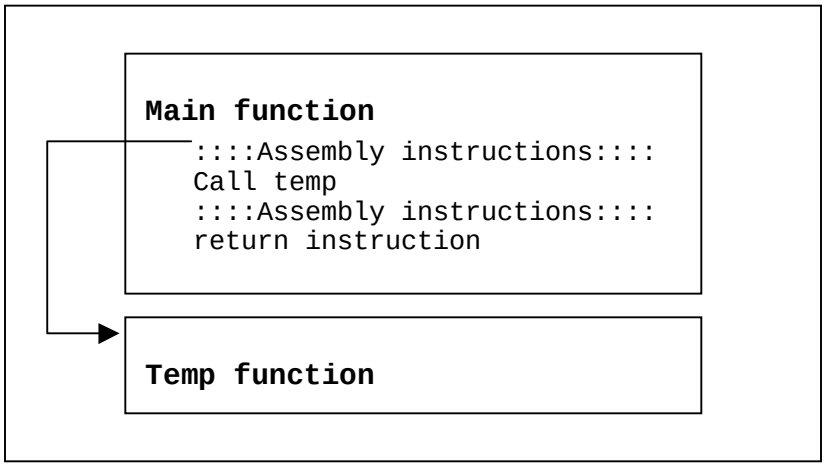


Figure 12.4: Structure of the First Version Assembly Language Program

During the second phase, the programmer initiates the IDE and, by pressing the F6 key, opens in the first version of the program a special mode known as the *hot swapping mode*. He or she can now edit the program.

For the sake of this demonstration, assume that the programmer has modified the functionality as shown in Figure 12.5.

The programmer has modified the `temp()` function in such a way that it displays numbers from 1000 to 5000 in increments of 1000. After the modification, the programmer presses the F1 key to compile the program.

```
// Main function
Void main(void)
{
      while(1!=2)
      {
            temp();
      }
}

// This function displays numbers from 1 to 5 whenever called
void temp(void)
{
      int a,b;
      a=1000; // Modified from the earlier version
      b=1;
      while(a<=5000) // Modified from the earlier version
      {
            cout a;
            a=a+1000;
            // The following loop reduces the speed of execution
            while(b<=1000000)
            {
                  b=b+1;
            }
            b=1;
      }
}
```

Figure 12.5: Second Version Program

After the IDE recognizes that the programmer has modified the `temp()` function, it generates the equivalent assembly instructions, as shown in Figure 12.6.

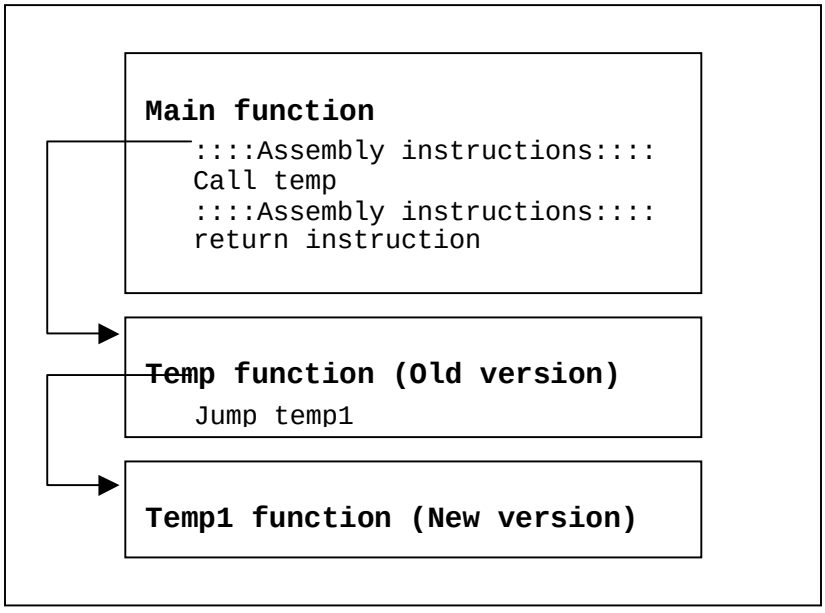


Figure 12.6: Structure of the Second Version Assembly Language program

Along with the assembly program, the IDE generates a new *make* file that instructs the NASM to execute in an appropriate manner. The NASM generates a listing of all the machine instructions and then the IDE reads those present within the list file of the new version of the `temp()` function and the `jump temp1` instructions, which are then copied into a hot pack.

The NASM generates in relative addressing code in such a way that the first instruction of the code starts from the location 0x00000000 (32 bit addressing). When this executable program is loaded into the memory, the address of the first instruction (0x00000000) is added with a situation specific offset, thereby producing a corresponding absolute address. The same calculation is extended for all other machine instructions of the executable file. For example, assume that the function `temp()` starts at a location 0x0000AAAA within the executable file. During the loading process, if the text area starts at a location 0x00001111, the absolute address of the `temp()` function will be calculated as 0x0000BBBB (0x0000AAAA + 0x00001111).

The hot swapper uses the same method to generate the absolute addresses for all the machine instructions present in the hot pack, which means that it must know the starting address of the text area. To accomplish this, the author of this program implements a preamble code that is executed whenever a target process is initiated. This code gets the process id by calling the `getpid()` system call, which returns the process id of the process from which the call was placed. Then, the preamble code copies the starting address of the text area into the accumulator register, an action that the simple instruction `MOV eax, main` can perform. The location of the `main()` function is copied into the `eax` register and then both the process id and the offset are stored in a data file known as the `runenv`.

The hot swapper first reads the `runenv` file and collects the process id and the offset. It then issues a `ptrace` system call and passes the process id. Based on the process id, the operating system suspends the execution of the first version of the target process. The hot swapper can then use the `ptrace` call to access the text area of the suspended target. It then adds the offset to the addresses of the machine instructions to produce *absolute addresses*. The hot swapper copied all the new machine instructions into their respective absolute addresses. Once all machine instructions are copied onto their associated absolute addresses, the process resumes execution — and the prototype hot swapper has successfully performed a hot swap.

Chapter 13: Conclusion

13. Summary

Since the programmer implements the functionality of an executing process within the executable code of a process's address space, and since during execution such code is tightly coupled with the associated instantaneous state, any ill-timed code or state update initiated by the programmer can cause incompatibility between code and state. To ensure compatibility, such modifications must be timed correctly; to ensure efficiency, it must also be possible for the dynamic modification process to perform them "on the fly." Dynamic modification is the concept of updating the binary image of an executing process without halting it. *Software Hot Swapping* is such a dynamic modification technique.

Section 13.1 reports on the contributions of this research, while Section 13.2 provides an outline of future work.

13.1 Contributions

Dynamic modification of an executing process involves any of the following modifications:

- Code modifications,
- Type modifications, or
- Data structure modifications.

Whenever type and data structure modifications are required, the dynamic modification process must also modify code. Hence an efficient code modification mechanism is crucial. The programmer develops code I(in the form of grains) using the constructs the underlying language or grains provide, and this research presents a hierarchy based on the conceptual sizes of such grains and reports on the effects of dynamic modification across this granular hierarchy. Based on this report, this research makes five contributions to the study of this subject.

The ***first contribution*** consists of categorizing the state incompatibilities into type incompatibilities, internal semantic incompatibilities, and external semantic incompatibilities. All previous researchers consider the state incompatibility as a combined hurdle. This author has examined the causes of these state incompatibilities and come up with a categorization that defines this hurdle more precisely..

The ***second contribution*** involves introduction of the integral section concept. The dynamic modification process must not modify a grain if the execution pointer is within the scope of its associated integral section. Based on the integral section concept, all the dynamic modification techniques can be theoretically classified as either of the two modification philosophies: *explicit safe spot mechanism* or *implicit safe spot mechanism*.

This author has chosen implicit safe spot mechanism as the design philosophy behind software hot swapping, because this philosophy is independent of the software architecture of the target application.

The ***third contribution*** of this research is construction of the integral section based on certain computable aspects and certain aspects that are undecidable. This author proposes a special data flow analysis algorithm, called the *integral semantic algorithm,* to compute aspects of the integral section. This algorithm first develops the control flow graph of the target program and maps the dataflow. The feasibility of this graph has already been demonstrated [EX 18]. This author also proposes a set of interactive to tackle the undecidable aspects of the integral section construction, which are due mainly to the nature of external semantic incompatibilities. The combination of the integral semantic algorithm and the interactive procedures together contribute to the construction of the integral section. The working of the integral semantic algorithm is based on reasonable informal logic. The author has reasonable belief that this integral semantic algorithm helps the programmer in the construction of the integral sections.

The ***fourth contribution*** entails proposing a set of characteristics that provide the basis ofthe design for any dynamic modification technique. This author proposes the implementation issues behind the feasibility of dynamic modification across the granular hierarchy.

The ***fifth contribution*** is the identification of the techniques involved in modifying the types and the data structures.

13.2 Future Work

This research uses a semi-formalized manner to present its concept of Software Hot Swapping. Several components, however, require extensive formalization, which is left for future research.

For example, the integral section concept requires such formalization. The concepts of feasibility and flexibility of dynamic modification, as explained in Chapter 7, Sections 7.3 and 7.4, also need to be formalized, as does the hot swapping calculus. One of the objectives of this research is to introduce an effective communication mechanism between the programmer and the hot swapper. The programmer-anticipated extensions concept requires formalization as well.

Detecting internal semantic incompatibilities involves data flow analysis. The integral semantic algorithm performs this verification and reports all the potential internal semantic incompatibilities and dictums that solve them to the programmer, who must then validate all the dictums that the system proposes. To do this, the programmer must verify the data flow graph. Therefore, the programmer needs a visualizer that displays the data flow analysis in a user friendly way. The nature of this visualization module and the way in which the data flow can be visualized are subjects for future exploration.

One of the conclusions reached by this research is that the programmer has to detect external semantic incompatibilities. At times, this detection activity can add a significant burden to the programmer's efforts. This author envisions an artificial intelligence-based expert system that advises the programmer of potential external semantic incompatibilities based on grain modifications. This expert system needs further exploration.

The hot pack development mechanism, described in Chapter 10, is an IDE-based development environment. Unix programmers also desire a VI-GNU C kind of development environment. This environment still needs to be designed.

Finally, this research concentrates on a single threaded program developed in an imperative language, like C. Software hot swapping has to be extended to an object oriented paradigm and a multi-threaded environment. Such extension requires further research and exploration.

Glossary

Active Grains
Those grains that have an associated state when runtime grain analysis is performed.

Dynamic Modification
Modifying the functionality of executing software without the need to halt its execution.

Dynamic Modification Event
The event at which the hot swapping functionality suspends the execution of the target functionality.

Executing Grain Set
A set of grains of unique sizes that holds within it the address associated with the instruction counter.

External Semantic Incompatibility
A situation that occurs when the executable code becomes incompatible with the external state.

External State
The set of all data stored within the operating system's address space that cannot be directly accessed by the executable code of the corresponding process that generated the data in the first place.

Function Call Graph
A graph constructed by considering all the functions as nodes and all the function calls as edges.

Grain
A term used to represent a language construct.

Inactive Executable Code
What the executable code becomes if the address associated with the instruction counter is not within the domain of a particular executable code.

Instantaneously Unused Grain (IUG)
What a grain becomes if the address associated with the instruction counter is outside the domain of its integral section.

Integral Section
A set of all contiguous regions in the control flow graph, within which the instruction pointer must not be present when dynamic modification is performed on the original grain.

Interactive Set
A set containing all identifiers that interact with the corresponding identifier under observation.

Internal Semantic Incompatibility
A situation that arises when unique grains of different versions of the same semantic set interact with the data flow during the dynamic modification process.

Internal State
The data that can be directly accessed by at least one machine instruction of the executable code.

Process Halting
Method by which an executing process is terminated forever with the assumption that to re-execute that process, the user must start from the beginning with new data and input.

Process suspension
Method which stops an executing process temporarily by retaining all its address space in such a way that it can resume execution from the same point whenever desired.

Run Time Grain Analysis
A test that categorizes the grains present within the address space of an executing process.

Safe Section
The region where it is safe to dynamically modify the associated grain. The safe section constitutes the entire address space, excluding the integral section.

Semantic Set
The set of all grains that can access at least one identifier within a data flow graph.

Suspended Grains
What a grain becomes if the address associated with the instruction counter is within the domain of its integral section.

References

Online Update Systems [OUS]

[1] M.E. Segal and O. Frieder, "On-the-Fly Program Modification: Systems for Dynamic Updating," *IEEE Software*, Mar. 1993, pp. 53-65.

[2] I. Lee, *DYMOS: A Dynamic Modification System*, doctoral dissertation, Dept. Computer Science, Univ. of Wisconsin, Madison, Wis., 1983.

[3] B. Liskov, "Distributed Programming in Argus," *Comm. ACM*, vol. 31, no. 3, Mar. 1988, pp. 300-312.

[4] T. Bloom, *Dynamic Module Replacement in a Distributed Programming System*, doctoral dissertation, Laboratory Computer Science, Massachusetts Inst. of Technology, Cambridge, Mass., 1983.

[5] M.E. Segal and O. Frieder, "On Dynamically Updating a Computer Program: From Concept to Prototype," *Journal of Systems and Software*, vol. 14, no. 2, Feb. 1991, pp. 111-128.

[6] D. Gupta, P. Jalote, and G. Barua, "A Formal Framework for On-line Software Version Change," *Trans. Software Eng.*, vol. 22, no. 2, Feb. 1996, pp. 120-131.

[7] D. Gupta, *On-line Software Version Change*, doctoral dissertation, Dept. Computer Science, Indian Inst. of Technology, Kanpur, India, 1995.

[8] M. Hicks, J.T. Moore, and S. Nettles, "Dynamic Software Updating," *ACM PLDI*, Snowbird, Utah, May 2001, pp. 13 – 23.

[9] M. Hicks, *Dynamic Software Updating*, doctoral dissertation, Dept. Computer Science, Univ. of Pennsylvania, Philadelphia, Penn., 2001.

[10] M.S. Day, *Replication and Reconfiguration in a Distributed Mail Repository*, master's thesis, Laboratory Computer Science, Massachusetts Inst. of Technology, Cambridge, Mass., 1987.

[11] J. Magee, J. Kramer, and M. Sloman, "Constructing Distributed Systems in Conic," *IEEE Trans. Software Eng.*, vol. 15, no. 6, June 1989, pp. 663-675.

[12] B.R. Rowland and R.J. Welsch, "The 3B20D Processor & DMERT Operating System: Software Development System," *Bell Systems Technical Journal*, vol. 62, no. 1, part 2, Jan. 1983, pp. 275-289.

[13] H.Goullon, R. Isle, and K. Lohr, "Dynamically Restructuring in an Experimental Operating System," *IEEE Trans. Software Eng.*, vol. 4, no. 4, July 1978, pp. 295-304.

[14] M. Dmitriev, *Safe Class and Data Evolution in Large and Long-Lived Java Applications,* doctoral dissertation, Dept. Computing Science, Univ. of Glasgow, Glasgow, Scotland, 2001.

[15] R.S. Fabry, "How to Design Systems in Which Modules Can Be Changed On the Fly," *Proc. 2nd Int'l. Conf. Software Eng.*, Los Alamitos, Calif., 1976, pp.470-476.

[16] R.S. Fabry, "Capability Based Addressing," *Comm. ACM*, vol. 17, no. 7, July 1974, pp. 403-412.

[17] M.R. Barbacci, D.L. Doubleday, and C.B. Weinstock, "Application-level Programming," *Proc. IEEE International Conference on Distributed Computing Systems*, Paris, June 1990, pp. 458-465.

Shared Libraries [SL]

[1] M. Franz, "Dynamic Linking of Software Components," *Computer*, IEEE CS, vol. 30, no. 3, Mar. 1997, pp 74-81.

[2] J.Q. Arnold, "Shared Libraries on UNIX System V," *Winter Conference Proceedings*, USENIX, Denver, 1986, pp. 395-404.

[3] Robert A. Gingell et al., "Shared Libraries in SunOS," *Summer Conference Proceedings*, USENIX, Baltimore, 1989, pp. 39-49.

[4] W.W. Ho and R.A. Olsson, "An Approach to Genuine Dynamic Linking," *Software – Practices and Experience*, vol. 21, no. 4, 1991, pp. 375 – 390.

[5] F.J. Corbato and V.A. Vyssotsky, "Introduction and Overview of the Multics System," *Proc. AFIPS Fall Joint Computer Conf.,* vo. 27, part 1, Spartan Books, Washington, D.C.,1965, pp. 185-196.

[6] M. Sabatella, "Issues in Shared Library Design," *Summer Conference Proceedings,* USENIX, Anaheim, Calif., 1990, pp. 11-23.

[7] G. Hjalmtysson and R. Gray, "Dynamic C++ Classes: A Light Weight Mechanism to Update Code in a Running Program," *Annual Technical Conference,* USENIX, New Orleans, June 1998, pp. 65-82.

[8] S. Liang and G. Bracha, "Dynamic Class Loading in the Java Virtual Machine," *OOPSLA,* ACM, Vancouver, Canada, Oct. 1998, pp. 36-44.

[9] C.B. Downing and F. Farance, "Transparent Implementation of Shared Libraries," *UniForm Conference Proceedings*, USENIX, Washington, D.C., 1984, pp. 209-221.

[10] L. Deller and G. Heiser, "Linking Programs in a Single Address Space," *Annual Technical Conference*, USENIX, Monterey, Calif., June 1999, pp. 283-294.

[11] W.W. Ho, W-C, Chang, and L.H. Leung, "Optimizing the Performance of Dynamically-Linked Programs," *Technical Conference Proceedings*, USENIX 1995, New Orleans, Jan. 1995, pp.16-20.

[12] L. Presser and J.R. White, "Linkers and Loaders," *Computing surveys*, ACM, vol. 4, no. 3, Sept. 1972, pp. 149-167.

[13] J. Levine, *Linkers & Loaders*, Morgan Kaufmann Publishers, San Francisco, 2000.

[14] M.A. Linton and R. Quong, "Linking Programs Incrementally," *TOPLAS*, ACM, vol. 13, no. 1, Jan. 1991, pp. 1-20.

[15] R. Quong, *Incrementally Linking*, doctoral dissertation, Dept. Computer Science, Stanford Univ., Palo Alto, Calif., 1991.

[16] S.M. Dorward, R. Sethi, and J.E. Shopiro, "Adding New Code to a Running C++ Program," *C++ Conference Proceedings,* USENIX, San Francisco, 1990, pp. 279-292.

[17] J.O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, Mass., 1994.

Hardware [HW]

[1] T. Shanley, *Pentium Pro And Pentium II System Architecture*, Addison-Wesley, Reading, Mass., 1997.

[2] D. Keppel, "A Portable Interface for On-The-Fly Instruction Space Modification," *Proc. 4th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, Santa Clara, Calif., 1991, pp. 86-95.

Zurich [ZU]

[1] D.G. Foster, "Separate Compilation in a Modula-2 Compiler," *Software Practice and Experience*, vol. 16, no. 2, Feb. 1986, pp. 101-106.

[2] L.B. Geissmann, *Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith*, doctoral dissertation, 7286, Dept. Computer Science, Univ. of Zurich, Switzerland, 1983.

[3] N. Wirth, "Type Extensions," *TOPLAS*, ACM, vol. 10, no. 2, Apr. 1988, pp. 204-214.

[4] G. Wirth, *Project Oberon*, ACM Press, New York, 1992.

[5] J.G. Mitchell, W. Maybury, and R. Sweet, *Mesa language manual*, Xerox PARC CSL-79-3, Palo Alto, Calif., Apr. 1979.

Reflection [RE]

[1] P. Maes, "Concepts and Experiments in Computational Reflection," *OOPSLA,* ACM, Orlando, Fla., Oct. 1987, pp. 147-155.

[2] N. Medvidovic, "ADLs and Dynamic Architecture Changes," *SIGSOFT,* ACM , San Francisco, 1996, pp. 24-32.

[3] P. Oreizy et al., "An Architecture-Based Approach to Self Adaptive Software," *Intelligent systems,* IEEE, vol. 14, no. 3, May/June 1999, pp. 54-62.

Software Engineering [SE]

[1] D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. ACM*, vol. 15, no. 2, Dec. 1972, pp. 1053-1058.

[2] D.L. Parnas, "Information Distribution Aspects of Design Methodology," tech. report, Dept. Computer Science, Carnegie-Mellon Univ., Pittsburg, 1971.

[3] D.L. Parnas, "A Technique for Software Module Specification with Examples," *Comm. ACM*, vol. 15, no. 5, May 1972, pp. 330-336.

[4] E. Perratore et al, "Fighting Fatware," *Byte* Apr. 1993, pp. 98-106.

[5] N. Wirth, "A Plea for Lean Software," *Computer*, CS IEEE, vol. 28, no. 2, Feb. 1995, pp. 64-68.

[6] M.D. McIlory, "Mass Production of Software Components," *Software Eng.*, P. Naur and B. Randell, eds., NATO Science Committee Report, Garmisch, Germany, Oct. 1968, pp. 138-155.

[7] M.A. Linton and R.W. Quong, "A Macroscopic Profile of Program Compilation and Linking," *Trans. Software Engineering,* IEEE, vol. 15, no. 4, Apr. 1989, pp. 427-436.

[8] E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, Mass., 1995.

[9] G.C. Necula, "A Scalable Architecture for Proof-Carrying Code," *5th Int'l Symp. Functional and Logic Programming*, Waseda Univ., Tokyo, Mar. 2001.

[10] F. DeRemer and H.H. Kron, "Programming-in-the-Large Versus Programming-in-the- Small," *Trans. Software Eng.,* IEEE, vol. 2, no. 2, June 1976, pp 80-86.

[11] B.W. Beach, "Connecting Software Components with Declarative Glue," *Proc. 14th Int'l Conf. Software Eng.* ACM, Melbourne, Australia, 1992, pp. 120-137.

Examples [EX]

[1] M. Gfeller, "Walks into the APL Design Space," *APL Quote Quad*, ACM, vol. 23, no. 1, July 1992, pp. 70-77.

[2] S.I. Feldman, "Make – A Program for Maintaining Computer Programs," *Software Practice and Experience,* vol. 9, no. 4, Mar. 1979, pp. 255-265.

[3] M. Blume and A.W. Appel, "Hierarchical Modularity," *Trans. Programming Languages and Systems,* ACM, vol. 21, no.4, July 1999, pp. 813-847.

[4] E .W. Dijkstra, "The Structure of 'THE' Multiprogramming System," *Comm. ACM*, vol. 11, no. 5, May 1968, pp. 345-346.

[5] S. Chiba, "Open C++ Tutorial," http://www.hlla.is.tsukuba.ac.jp/~chiba/openc++.html, (Last Updated: Oct. 17, 2001).

[6] A. Mohindra and M. Deverakonda, "Dynamic Insertion of Object Services," *COOTS,* USENIX, Monterey, Calif., June 1995, pp. 13-20.

[7] D. Decasper et al. A Software Architecture for Next Generation Routers," *Conf. Applications, Technologies, Architectures, and Protocols for Computer Communication,* ACM *SIGCOMM*, Vancouver, Canada, 1998, pp. 229-240.

[8] R. Ierusalimschy, L.H. de Figueiredo, and W.C. Filho, "Lua-an extensible extension language," *Software Practice and Experience,* vol. 26, no. 6, Oct. 1994, pp. 635-652.

[9] M. Franz and T. Kistler, "Slim Binaries," *Comm. ACM,* vol. 40, no 12, Mar. 1997, pp. 87-94.

[10] C. Hastings, Jr., J.T. Hayward, and J.P. Wong, Jr., *Approximations For Digital Computers*, Princeton Univ. Press, Princeton, N.J., 1955.

[11] R.F. Rey, ed., *Engineering and Operations in the Bell system*, 2nd ed., AT&T Bell Laboratories, Murray Hill, N.J., 1986.

[12] B. Plattner and J. Nievergelt, "Monitoring Program Execution: A survey," *Technical Conf. UNIX and Advanced Computing Systems,* USENIX, vol. 8, no. 2, 1995, pp. 107-133.

[13] T. Adams et al., "Sustainable Infrastructures: How IT Services Can Address the Realities of Unplanned Downtime," *Gartner Dataquest Group*, Gartner, May 2001.

[14] H. Lingsong, "SVGA Graphic APIs," http://heliso.tripod.com/dosapis/graphic/graphic.htm (Last Updated: Oct. 1, 2000).

[15] Netwide Assembler Documentation, http://sourceforge.net/forum/forum.php?forum_id=167171 (Last Updated: Apr. 8, 2002).

[16] M. Beck et al., *Linux Kernel Internals*, 3rd ed., Addison-Wesley, Reading, Mass., June 2002.

[17] S. Maxwell, *Linux Core Kernel Commentary,* CoriolisOpen Press, Scottsdale, Ariz., 1999.

[18] R. Johnson and K. Pingali, "Dependence-based Program Analysis," *PLDI*, ACM, vol. 28, no. 6, June 1993, pp. 78-89.

[19] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers Principles, Techniques and Tools*, Addison-Wesley, Reading, Mass., 1988.

[20] *The Sun Enterprise Cluster Architecture*, Technical White Paper, Sun Microsystems Inc., Mountain View, Calif., 1997.

[21] P.L. Anderson and G.C. Anderson, *Advanced C: Tips and Techniques*, Hayden Books, Indianapolis, Ind., 1989.

[22] G.M. Hopper, "The Education of a Computer," *Proc. ACM Conf.*, ACM, May 1952, reprinted *IEEE Ann. of History of Computing*, ACM, vol. 9, no. 3, July/Sept. 1987, pp. 271-281.

Vita

Pradeep Tumati was born in Kakinada, India in 1977. When he was in grade 8, his father gave him a PC. He took a fancy to it, learned BASIC with the help of the self-manual, and exhibited a program called "**Operation Gulf War I**" at the school day function in date?. All his teachers and classmates congratulated him. That event inspired him to make computers his career and himself unique and important in that field.

Tumati began his undergraduate studies at the University of Madras in August 1995. When he was an undergraduate, he worked as a part-time freelance software consultant. The experience he gained working on a few software projects of varying degrees of complexity enabled him to obtain an application software development contract from "M/S CHITALE & SON," a leading architecture firm, after many demonstration sessions persuaded the management that the firm needed his skills. He successfully completed the assignment to the company's satisfaction and earned a handsome remuneration, which financed his education for an year.

Tumati received his Bachelor degree in Computer Science and Engineering from the University of Madras in May 1999 and, in the fall of that year, enrolled in the graduate program of the Computer Science Department at Virginia Tech, Blacksburg, VA. During his first semester, he decided to pursue research under Dr. John (JAN) Lee in the area of software hot swapping. After researching this idea for about an year, he and Dr.Lee realized that this technology could be marketed commercially. Since then he has been in the process of starting a company in the Silicon Valley. His interests are in programming languages, dynamic software modification, and operating systems. He graduates with a Masters of Science degree in Computer Science in May 2003.