# A FRAGMENT-BASED PROGRAM EDITOR

Surajit Choudhury
School of Computer Science
McGill University, Montreal

August 1986

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

## Abstract

Integrated software development environments are assuming considerable importance in the task of developing and maintaining medium to large scale software. Central to such environments is an editor which has knowledge of the syntax and semantics of a particular programming language.

This thesis presents the design of an editor within an integrated environment that allows programming in fragments. Fragments are independent structural components of software and their usage is an attempt to promote software reusability. Two primary implementation issues directed the design presented in this thesis - portability and adaptability. A system architecture is presented that encapsulates the potentially non-portable components within well-defined modules. Each editing operation requires enforcing the language rules. A structure classification scheme is presented for a table-driven implementation with quick resolution of the syntax rules and adaptability. The importance of incremental handling of semantics within such editors is documented and a simple to implement approach is presented, along with the special support for fragments.

## Résumé

Les environnements intégrés pour le développement de logiciel ont une importance grandissante dans le domaine du développement et de l'entretient de logiciel de moyenne à grande échelle. L'élément essentiel d'un tel environnement est un éditeur ayant connaissance de la syntaxe et de la sémantique d'un langage de programmation donné.

Cette thèse présente le "design" d'un éditeur pour un environnement intégré qui permet la programmation en fragments. Les fragments sont des éléments structurels indépendents de logiciel, et leur utilisation est un essai pour promouvoir la réutilisation de logiciel. Deux points d'importance ayant trait à l'implémentation ont guidé le "design" présenté dans cette thèse: la portabilité et l'adaptabilité. Une architecture du système qui isole les éléments potentiellement non-portable à l'intérieur de certains modules précis, est présentée. Chaque opération de l'éditeur nécessite une stricte adhérence aux règles du langage. Une méthode de classification des structures pour une implémentation utilisant des tables pour une résolution rapide des règles de syntaxe et pour faciliter l'adaptabilité est présentée. L'importance de manipuler la sémantique du langage de façon incrémentale pour de tels éditeurs est documentée, et une méthode d'implémentation simple, avec le support spécial nécessaire pour les fragments est présentée.

## Acknowledgements

This thesis has been completed with the support and help from numerous individuals and organizations. I would like to thank my supervisor Dr. Nazim Madhavji for his academic guidance throughout the course of this research. I am grateful for the numerous times that I had turned to him for support, and he was willing to help. It is a pleasure to acknowledge the company of the other members of the research project - Luc Pinsonneault, Rob Robson and Kamel Toubache. Part of this research had been made possible through the financial support of an NSERC, Canada research grant. Financial support during my graduate studies had also been offered by the School of Computer Science through teaching assistantships. I gratefully acknowledge the support from both these organizations. Finally, none of this would have been possible had it not been for the love and encouragement from my parents.

# Table of Contents


**Abstract** ..... i

**Résumé** ..... ii

**Acknowledgements** ..... iii

Chapter 1: **Introduction** ..... 1

1.1 Language Knowledge in Editors ..... 2

1.2 Software Reusability and Program Fragments ..... 3

1.3 Issues in Implementation ..... 4

1.4 Related Work ..... 6

Chapter 2: **The MUPE-2 System** ..... 9

2.1 Software Development and Modula-2 ..... 10

2.2 Programming in Fragments ..... 13

2.3 The User Interface ..... 16

2.4 The Command and Response Language ..... 18

Chapter 3: **Architecture of the Editor** ..... 22

3.1 The Screen Manager ..... 23

3.2 The General Manager ..... 25

3.3 Single Processor GM-SM Communication ..... 31

3.4 Extensions to a Distributed System ..... 35

Chapter 4: **A Scheme for Enforcing Syntax Rules** ..... 37

4.1 Types of Language Based Editors ..... 38

4.2 Editing Operations ..... 39

4.3 Structure Classification ..... 42

4.4 Table-driven Menu Generation and Compatibility Checking ..... 49

Chapter 5: **Contextual Issues in a Fragment-based Program Editor** ..... 52

5.1 The Incremental Nature of the Problem ..... 52

5.2 The Attribute Grammar Approach ..... 54

5.3 The Identifier Map Approach ..... 57

5.4 Incomplete Information in Fragments ..... 62

5.5 Interfragment Operations and Dependencies ..... 65

5.6 Issues in Interface Control ..... 68

**Conclusions** ..... 72

References ..... 75

# Chapter 1: Introduction

Programming methodology in recent years has seen a large number of research efforts and its importance in the industry has increased considerably. This has risen from the fact that software systems are evolving into increasingly larger and more complex entities than their predecessors. Consequently, the tools and techniques for programming of a decade ago do not provide the levels of software productivity that are expected in developing today's systems. An approach to improve software productivity is the reuse of software within a particular application area.

Concurrent to the changing nature of software systems, there have been new developments in *Software Development Environments*. The primary emphasis has been that they should provide increased support to the evolving nature of programming methodologies. As a result, such environments are providing, among others, language support through all phases of software development, and an integrated concept in the user-computer communication. A very familiar tool to the software developer is the Editor. Thus the editor has been the focus of attention for evolution, or even revolution, in the advanced environments.

The purpose of this thesis is to discuss the issues involved in the design and implementation of a Fragment-based Program Editor and to present some original research contributions in this area. This Fragment-based Program Editor is an attempt to address the increased demand on software productivity due to the increase in size and complexity in the requirements of present-day software systems. It combines recent developments in language-based program editors with the methodology of software reusability. A system architecture for such an editor is presented in this thesis that is derived from adaptability and portability requirements. Another major contribution of this thesis is a structure classification scheme for encoding language knowledge in editors.

Incremental semantic checking algorithms are studied and a new approach, made necessary by the interface control properties of Modula-2, is presented.

## 1.1. Language Knowledge in Editors

In a typical conventional programming environment, there are two major tools that a software developer utilizes most of the time. One is the text editor that is used to enter and modify programs written in a programming language, and the other is the compiler, which translates the user's specification to the abstract machine, the program, into a form that can be executed. In many cases, the environment would provide a number of compilers, and thus allow programs to be written in a number of programming languages. A common editor is used for entering and modifying programs written in the many languages for which compilers are available in the environment. Consequently the interactive editor is language independent, and it is the role of the compiler to inform the user of any language errors in the program.

However, the independence between these two major tools leads to less than desirable user support in software development. In particular, this is due to the presence of what has been termed as the 'edit-compile' cycle. It becomes necessary to repeatedly edit, scan and parse a complete program, if only to obtain an indication of absence of syntax/semantic errors. Such a situation can be avoided by incorporating language rules in the editor and in effect integrating the compiler functions with that of the editor's. Thus each user's editing operation is carried out in concordance with, or post-validated by the language rules. This makes it possible for the user to be aware, within the editor, if there is any conflict between the program being edited and the rules of the language.

Further benefits also arise with such an integrated interactive tool. It becomes possible to utilize unused computer time to incrementally generate code for the program being edited. It is also possible to provide instructions to the user by indicating what changes are permitted at different parts of the program. Further, the editor can provide immediate program formatting while in the editor, which results in a feedback to the user for program semantics. In particular, for languages with the 'dangling else' problem, this

feedback can be helpful. In general, in a highly integrated programming environment, the 'editor' provides all the interface. This results in an uniform user interface, a 'modeless' environment and consequently an environment which can potentially be easy to learn and simple to use.

## 1.2. Software Reusability and Program Fragments.

Software reusability holds the potential of providing significant improvements in software productivity, as well as software quality. The reusability in software applies not only to reusability in pieces of program code, but also to reusability in development knowledge, domain knowledge, among others. However to accomplish these many goals of software reusability requires increased support to the user It is expected that the concept of *program fragments* will provide a foundation on which an environment promoting software reusability can be constructed.

Program fragments are building blocks of a software system. These blocks can exist in a number of forms. They could be a few program statements developed and used by a single individual. They could be a set of data structures that are used to implement a frequently used data type. They could be program 'modules' that operate as servers of resources within a software system. Thus program fragments offer a multitude of interpretations and are different from the prevalent concept of modules. A software module arises from the need of the designer to decompose a system into functional units, to impose hierarchical ordering on function usage, to isolate machine dependencies, or to ease debugging, testing, integration, tuning, and modification of the system Modularization is directed at being a top-down development The criteria used for modularization include information hiding, loose coupling among the modules, matching the modular structure of the system with that of the problem.

On the other hand, software development through exclusive usage of large granularity objects, such as modules, does not extend to providing an improved software development methodology to the individual programmer. A requirement for the individual programmer in exercising software reusability is the support of *incomplete* program

fragments. Such program fragments are typically language structures of fine granularity, or are in an 'unrefined' form. Thus the incompleteness of these program fragments arises because they may consist of only data declarations, without a program body, or a fragment representing only informal program specifications. None of these types of fragments is strictly the 'compilation unit' that a programming language recognizes as an individual entity.

However in the software development process, it often becomes necessary to apply a mixture of top-down and bottom-up approaches. A fragment of data declarations may first be built and then successively tested with different algorithms before the programmer chooses one for the application being developed. The partial systems that are developed in this process are likely to be useful for some other applications. It would be to the programmer's benefit if it were possible to store the software components with some amount of design information to make the process of subsequent interconnection easier. Furthermore, the software development environment should be aware of the 'incompleteness' of the partial systems and also possess language knowledge in providing an integrated environment, as mentioned earlier.

A Fragment-based Program Editor allows such an environment to be provided to the individual programmer. The editor possesses knowledge of the language rules and can thus *guide* the development of programs that abide by these rules, within the editor. The editor promotes software reusability by recognizing software components of a wide spectrum of granularity in the language, as individual entities. It treats structures of all granularity in an uniform manner and it is driven by a set of *Fragment Construction Rules* that maintains the construction of fragments to be valid structures as per the language's syntax rules.

### 1.3. Issues in Implementation.

In the design of complex software systems, it is evident that such systems may require transfering to different software and hardware environments. A system that allows easy transfer is referred to as a portable system. A Fragment-based Program

Editor is one such system for which the portability goal is very relevant.

In a system like an editor, the functionality of the system is concentrated in a set of tools that are specific to the system. Thus, it becomes possible to isolate design decisions in separate subsystems which do not affect the specific tools, and consequently, the functional behavior of the system. Replacing such system components, as when the underlying hardware changes, would not affect the other system components It thus becomes necessary to develop a layered design in which lower layers provide facilities to the implementation of the functional tools at the upper layers. Two main components can be identified in the Editor to provide the desired tool independent facilities One is the *user interface system*, and the other is the *internal representation manipulation system*

The user interface system is responsible for the display of programs on the screen, the display of any menus, and the interpretation of input from the keyboard and any pointing devices. In many recent workstation computer systems such capabilities are offered within a window management system. However, these systems are not standardized and easy portability of the editor across such workstations should be aimed for A user interface system that encapsulates the window manager and input devices dependent features is a component at the lower layers of the design. Thus in porting the editor among a variety of display terminals and input devices, the necessary modifications are restricted to such an user interface system.

The Fragment-based Program Editor allows editing of *structures* These structures are elements of the language. Thus to permit efficient structure editing, the program fragments are maintained in structured form, mostly in tree forms. In such a case, the structure editing operations can directly translate to primitive operations like grafting and pruning on a tree. However the method of maintaining the structures is often dependent on the characteristics of the underlying system on which the implementation is being carried out. For example, in a system with an efficient data base system, such structures may be maintained by the data base management system. Thus the specific tools of the editor should be independent of the method of the internal representation of the structures. The internal representation manipulation system provides the needed

independence. It abstracts the structural information needed in deriving the contextual information necessary for driving the Editor, it provides the structure editing facilities, and it also provides the textual representation from the structures.

Apart from portability, another considerably important aspect to consider in any design is adaptability. Adaptability, and synonymously, extensibility, is concerned with the ability to make modifications of a system according to changing requirements. Within the functionality of a program editor, the foreseeable modifications are in changes to the language's rules, or changes arising with increased usage of the system, mainly with the user interface model.

The formalism inherent in language rules translates well to a concept that is particularly attractive for adaptability. This is the concept of *table-driven* implementation, and has been popular for generation of compilers. The application of the language rules in the Editor takes place at every editing operation. This results in indicating whether a structure editing operation is valid according to the language's rules. In a table-driven implementation, such validation can be performed by accessing a table with the editing command and the operands of the operation to obtain the truth value on the validity of the operation. Thus changes to the language rules can be accommodated by changing the contents of the tables, changing the range values on one dimension, or even changing the dimensionality of the table. Furthermore, it is possible to partition the tables so that changes in command semantics have localized effects.

Adaptability in the user interface design allows potential users to tailor the interface to their requirements without affecting the functionality of the system. This applies to adjusting the screen layout, or the color coding scheme in a graphics based interface; in a textual interface it could be the facility to alias and chain commands. The user interface system, as explained earlier for portability requirements, provides the basis for this adaptability requirement.

## 1.4. Related Work.

The improvement of the programming process has been the object of study by many research groups. Such efforts have resulted in the development of programming environments that support programmers in the process of transforming specifications into working programs. This process involves creation and modification of programs, checking their consistency, generation of an executable form, and monitoring of the program's behavior. An integrated programming environment seeks to provide a set of tools that share the program's representation, present a consistent user interface, and in cases, even provide the functionality of the environment encapsulated as a single tool

The early research in the area of programming environments is best represented by the Cornell Program Synthesizer[TeiRe81], Mentor[Donze80] and Interlisp [TeiMa81] The Synthesizer supports syntax-directed editing, execution, and debugging of programs in a subset of the PL/1 language. It is a well-integrated programming environment and it is primarily designed as a teaching tool. The original Mentor editor is a language-based editor for Pascal. While the Synthesizer mainly used templates to build programs, programs in the Mentor editor were entered as text, then parsed, and subsequently allowed structure oriented manipulations. Interlisp is an advanced programming environment for LISP. The system includes a number of highly integrated tools. It has been used in experimental systems characterized by prototyping and design iterations

Among the more recent research projects in language based environments are GANDALF[Notki85], Pecan[Reiss84a], Syned[Gasne83], and Magpie[DeIMS84] While the functionalities of these systems are not too different from those of the earlier generation, they have, in some cases, approached the idea of generating environments for different languages from a language independent generator that accepts the language rules. In others, the emphasis had been in extending the systems formerly suitable for, and sometimes restricted to, teaching purposes, to deal with realistically large software systems

The question of providing reusability within an integrated programming environment that is addressed in this thesis finds correspondence with very few research projects. The Programming System Generator[BahSn85] considers the basic unit of editing

and interpreting to be fragments that are arbitrary parts of a program. However, it does not consider the typing of fragments to derive the rules of editing, as has been done in the system considered in this thesis. The IOTA Programming System[NakYu83] supports type-parameterized modules as the basic unit of programming. While this appears to be the closest to the concept of typed fragments, the granularity of the modules in IOTA do not offer the same level of fineness.

This thesis consists of five chapters. The following chapter introduces the MUPE-2 system for which the Fragment-based Editor has been developed. Chapter 3 describes the architecture of the Editor paying attention to the requirements of portability and adaptability. The next chapter is the description of the scheme adopted to derive an adaptable language based editor. Chapter 5 discusses the handling of the language's contextual issues within the language based editor.

# Chapter 2 : The MUPE-2 System

The MUPE-2 (McGill University Programming Environment) project aims to provide the programmer with a system which will simplify the programming and maintenance processes of medium to large scale software projects. Towards this end the MUPE-2 project seeks to improve on the command language, feedback response, and protection issues, among others.

The primary feature of MUPE-2 is in providing the user the ability to program in *fragments*. Fragments are structural components of a program that exist independently in the environment. The scope of fragments is enlarged by considering documentation issues in addition to purely programs. Thus it is possible to have fragments that hold natural language descriptions - abstracts. The contents of a fragment decide the type of the fragment. This type, denoted the *fragtype*, determines the availability of operations, specification of options to commands, and the applicability of other fragments for an operation on a fragment. The fragtype of a fragment could potentially change in the development process. Such a feature practically allows the user to develop software in a uniform bottom-up and top-down manner.

The use of typed fragments in MUPE-2 allows an unification of the traditionally two distinct activities: programming in the large, and programming in the small Programming in the large normally refers to programs that are large, developed by a large group of people, and are meant to be used for a long duration of time To tackle the accompanying complexities in debugging, testing and modification, the large program is decomposed into modules. Thus *programming-in-the-large* activities are normally concerned with program units or modules, and the interrelationships between them [DeReK76]. However such activities have been strongly separated from programming in the small activities for which the use of syntax directed editors have normally been aimed

at. As a result such editors were restricted to activities like introducing variables, controlling the flow of control, and maintaining the basic internal static semantic consistencies.

The MUPE-2 system provides an uniform view of programming to the user. The above two activities are not distinguished because there is a common set of commands. The system allows the user to consider all activities alike, whether an activity involves operating on a single fragment or multiple fragments. The applicability of a command is decided upon by the structure on which the operation applies Further discussions on the programming view presented in MUPE-2 can be found in [Madha85].

## 2.1. Software Development and Modula-2

In designing a programming environment to support a programming language, it is evident that many of the environment's features are decided by the language and the model of software development that it encourages. The language for which MUPE-2 is designed is Modula-2 [Wirth85], and this section examines the model that is behind it and the facilities the system provides to the user.

Modula-2 is one of the more modern programming languages that has been designed with the intention of tackling the 'crisis in the software industry'. The language provides a reasonable balance between simplicity and functionality, although it is not without its criticisms [Powel83, MadPT86]. Nevertheless, it is a close member of the popular 'Algol family', which should prove the language to be easy to learn and use among the programming community. Nonetheless, Modula-2 includes many features that are beyond the scope of many of the earlier languages. The language's particular attraction lies in it being usable for both 'low-level' programming - for which assembly language was the only resort - and for developing large scale software systems.

The most distinctive features of Modula-2 are:

- Strong typing with static checking. As with its predecessor Pascal, Modula-2 employs a strong type checking for operand compatibility during compilation time.
- Separate specification of interfaces and their implementation. Each unit of compilation is composed of a definition and implementation module-pair. The definition

module specifies the interface of the compilation unit and the implementation module provides the implementation. The two modules exist as distinct textual entities.

- Data abstraction. It is possible to hide the type specification of user defined types Such types are accessible as opaque types, and are only relevant via the operations that are available to be applied on them.
- Systems programming. Access to machine specific details are possible due to the facility of allocating space for a variable at specific addresses in the memory map
- Concurrency Single processor quasi-concurrent operations are permissible by the concept of co-routines.

Another language with similar features that has attracted a lot of research interest recently has been the Ada programming language [Unite82]. It is particularly interesting to note the specifications of the Ada Programming Support Environment [BuxtD81] The primary reason for developing such specifications is that not only the language, but the environment for software development should also be portable. However no such development has been noted with Modula-2 and it is apparent that research in similar directions for Modula-2 is urgently needed. Languages such as Ada and Modula-2 are intended for implementing software systems that are large and complex. Such systems are normally composed of a large number of modules and as a result the support for developing these systems require tools that are different and beyond the (text) editors, compilers and debuggers that have so far been used in systems development

The design of complex systems often require a subdivision of the task into more manageable components that are within the scope and knowledge of a single specialist. Many issues are relevant in the decomposition, but the dominant theme is that the decomposition results in a system with components that are tightly contained within themselves and minimize the interaction with other components As a result the development of individual components can be handled in an independent manner Interactions between components is restricted to the functional usages of the components among themselves. This has also been interpreted as the individual components being producers

and consumers of resources. Any single component is composed of resources, and it is participating in the development of the total system by providing some or all of these resources for use by the other components. This is the basic idea behind decomposition of systems and the resulting notion of modules, or components.

Modula-2 addresses the software development process in the above discipline. A primary aspect in Modula-2's programming model is the concept of interface specification. Every module that is playing the role of a component, as explained in the previous paragraph, specifies its interface via 'import' and 'export' lists. A module's export list enumerates, in a purely syntactic manner what the module provides as resources contained within that may be accessed by others. In a similar manner, a module obtains access privileges to another module's exported resources by an import declaration. Since the export list of a module constitutes the interface specification that is of interest to other modules, Modula-2 allows this specification to be textually separate, that is to reside in separate files, from the rest of the module which describes its implementation. This means that it is possible to have independent development of modules since the interface control mechanism is directed only at the interface specifications The complete implementation of the modules are necessary only during the linkage editing phase for system integration.

With such a model of software development in Modula-2, it is apparent very soon that it does not get good help from conventional operating systems' tool-kit and file-oriented approach. Many of the command languages, for reasons of uniformity among multiple languages, are designed to work on a uniform structure, the file. However to consistently provide access to the structure embedded in any program written in Modula-2, the programmer should be provided with an environment that understands such a structure. MUPE-2 follows the recent trend in programming environments in providing an internal representation of programs structured by the rules of the language in which the program is written. These structural rules now dictate the scope and applicability of the user's operations within the environment However, what MUPE-2 seeks to improve upon similar environments that involve the usage of a language's structure, is to provide a better environment for the development of medium to large scale software

systems using Modula-2. The following section introduces the concept of fragments and their relevance in programming environments.

## 2.2. Programming in Fragments

The unit of program that is supported in MUPE-2 is termed a fragment. It is different from a program because it is more close to being a building block for a complete Modula-2 program, than what is achievable with the compilation units, or modules However it should be understood that fragments are not some form of sub-compilation units. Fragments can consist of a compilation unit and can also consist of a collection of such units. This provides for an integration of the distinct activities of programming with small structures and the activities associated in 'integration', or programming in the large. Consequently, the concept of fragments allows the introduction of a software development discipline that is uniform throughout the development process.

The primary element in this discipline of software development with the use of fragments is the concept of fragment types, or *fragtypes*. A fragtype associates the syntactic information content of a fragment with a scalar value that is used for providing the necessary rigor inherent in the language rules. The concept of fragtypes is general enough to be applicable to a wide variety of languages, but this thesis will discuss only its application to the programming language Modula-2.

In a program written in a high level programming language, one can identify component parts that combine to make up the whole program. It is possible to associate these component parts as building blocks in a software development process Furthermore, it is evident that there is a hierarchical structure in any program, as is apparent from the production rules defining the syntax of the language. Moreover, there is a finite number of classes which these building blocks can be classified into The classification values form the set of fragtype values. The fragtype values that MUPE-2 provides for programming in Modula-2 in its environment are as shown below.

The definition of fragtypes is conseqently based on the concept of software building blocks. In the following, a BNF-like notation is used to define fragtypes in terms of an

| | |
|---|---|
| Expression | Def-Imp Module |
| Declarations | Program Module |
| Statements | Unit Subsystem |
| Abstract | Procedure Subsystem |
| Unit | Module Subsystem |
| Procedure | Def-Imp Module Subsystem |
| Module | Program Module Subsystem |
| System-Layer | |

extended Modula-2 construct set. The symbol /|\ means *root-of*, and <..> means *inserted-around*. The rules are meant to be used in conjunction with the language rules of Modula-2, as found in [Wirth85]. There are loose classes among the various fragtypes representing structures of similar characteristics, and the fragtype definitions are presented by these classes for purposes of clarity.

| | | |
|---|---|---|
| **Procedure Subsystem** | ::= | Procedure /\|\ {Internal-node}$^1$ |
| **Module Subsystem** | ::= | Module /\|\ {Internal-node}$^1$ |
| **Def-Imp Module Subsystem** | ::= | Def-Imp Module /\|\ {Internal-node}$^1$ |
| **Program Module Subsystem** | ::= | Program Module /\|\ {Internal-node}$^1$ |
| **Unit Subsystem** | ::= | Unit /\|\ {Internal-node}$^1$ |

The above are the definitions for the several *subsystem* fragtypes. Each of these is identified by the type of construct that forms the root node of the subsystem. For example, **Program Module Subsystem** is the fragtype of a fragment that has a Program Module construct as the root of subtrees of Internal-nodes. Internal-nodes, for notational purposes, is defined as

Internal-node = Unit | Procedure | Module.

Procedure and Module relate to Modula-2 primitive constructs, while Unit is an *unrefined* construct of the previous two. Its usage is to allow deferring a choice between Procedure and Module, mainly in the early stages of system design. Thus a subsystem of

configuration A/|\(B,C,D) represents A as the root node of internal nodes B, C and D.

**System-layer** ::= {Internal-node}$^{2}$

**A System-layer** is defined as the fragtype of a fragment with at least two structures of type Internal-node. An example configuration is (C,D), where C and D are Internal-nodes.

| | | |
|---|---|---|
| **Procedure** | ::= | Procedure-template<..>Template-contents |
| **Module** | ::= | Module-template<..>Template-contents |
| **Unit** | ::= | Unit-template<..>Template-contents |
| **Def-Imp Module** | ::= | Def-Imp Module-template<..>Template-contents |
| **Program Module** | ::= | Program Module-template<..>Template-contents |

The above are the definitions for procedure and module fragtypes. Each of these is identified by the type of template that is inserted around its contents For example, **Module** is the fragtype of a fragment that has Module-template inserted around Template-contents. For notational purposes, Template-contents is defined as,

Template-contents = {Declaration | Phrase} {Statement | Phrase}

Declaration and Statement relate to Modula-2 primitive constructs, while Phrase is similar to a comment that can be refined to primitive constructs, depending on its context. Its usage is again similar to that of Unit, allowing system design at higher levels of refinement. As an example, Procedure-template<..> Declaration Statement is a fragment of fragtype **Procedure**.

| | | |
|---|---|---|
| **Statements** | ::= | {Statement}$^{1}$ |
| **Declarations** | ::= | {Declaration}$^{1}$ |
| **Expression** | ::= | Expression |
| **Abstract** | ::= | {Phrase}$^{1}$ |

**The above are the definitions of lower-level fragtypes. The associated fragments represent homogenous constructs. For example,**

```
a := 5;
b := 6
```

is a fragment of fragtype **Statements**. Fragments of fragtype **Abstract** undergo changes in fragtype value as the first Phrase in them is refined to any primitive construct of the language.

## 2.3. The User Interface

Significant advances in computer graphics technology have allowed system designers to experiment with new approaches to the computer-human interface. The most prominent developments have been the use of pop-up menus, pointing devices and windows. Menus have provided an effective interface because the user does not have to memorize command words. Fast and easy selection by pointing has opened a new era in command languages. The concept of windows has allowed the user to deal with several program 'views' on the screen at the same time. Windows also allow several jobs to be initiated and monitored simultaneously, thus leading to more effective usages of a multitasking computing environment.

The user interface for MUPE-2 has been particularly designed to take advantage of these new developments. Furthermore, the MUPE-2 research seeks to utilize colour as a mode of communication between the system and the user. A preliminary sketch of the MUPE-2 user interface is shown in Figure 2.1.

The screen is partitioned into three non-overlapping windows, or 'tiles'. These constitute the **Module Screen**, the **Scratch Pad** and the **Procedure Screen**. The Module Screen is used to view and edit in the hierarchy of nodes at the procedure or module level. The Procedure Screen allows the user to edit within a node. The nodes that are internally viewed and edited in this screen are the ones present in the node hierarchy viewed in the Module Screen. Multiple nodes from the Module Screen can be *opened* for editing in possibly overlapping windows within the Procedure Screen. The editing in these two screens are done in context of each other. In fact, the structures edited in these two screens are part of the compilation unit structure whose node hierarchy is viewed on

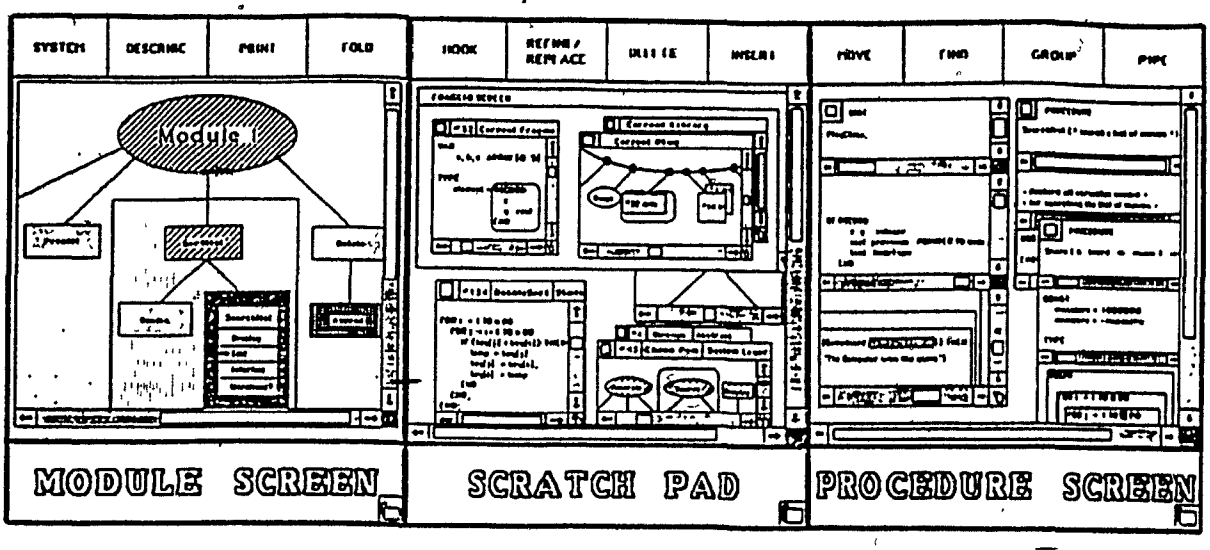


Figure 2.1 - The MUPE-2 Structured User Interface.

the Module Screen and individual nodes are viewed in the Procedure Screen. Thus the effects of editing in one of the screens is apparent in the other screen.

Unlike the previous two screens, the Scratch Pad allows the user to edit in a context-free setting. Within this screen it is possible to deal with program fragments such that they can be developed, refined and assembled out of context of a main program. The program fragments are of a wide spectrum of program granularity, from expressions to system layers. Multiple fragments can be *open* for editing in the Scratch Pad via the use of windows.

The top-most portion of the user interface on a graphics display provides a set of labeled 'buttons' that are selectable to invoke commands. It is a common set for the three screens. Since at any instant there is only a single active fragment, only the applicable subset of the commands for the designated structure within that fragment will be available. These are the commands that are available to the user at the editing cursor in

the active fragment. The available commands' associated buttons are highlighted at all times. The Scratch Pad region is also available to provide a view of the library of fragments, called Fraglib. This library is accessible from the three screens and access to and operations in the library are done with the same set of commands as used in editing.

In the GANDALF project [Notki85], similar ideas to the above may be found in their use of *scenes*. A scene works in coordination with multiple unparsing schemes and the window manager, to provide a better abstraction of the program tree. When a scene node is entered, a new window is opened up and the scene node becomes the root of that window. Procedure bodies elided via one unparsing scheme, are then focussed on via a different unparsing scheme, without being distracted by the remainder of the program. However the MUPE-2 design is closer to the three-screen user interface as has been used in the Spatial Data Management System [Herot80]. In that system, one screen provides a world view of the data base, which is a coarse index to the whole system. On the other two screens, the user can obtain an exploded view of a portion that is highlighted on the world-view screen.

## 2.4. The Command and Response Language

The human-computer interface design of a computer system is a difficult and complex undertaking which involves a wide range of considerations. The language of communication between the users and the computer system is often referred to as the Command and Response Language (CRL). The user communicates with the computer system by means of commands to utilize the set of services that are available in the system. Commands are accepted as input by the system and can cause the information stored by the system to be updated, or to produce output in the form of responses. Responses could provide the information requested by the command, or could be an indication of the state of the system.

A wide variety of CRLs are in use in present-day computer systems. While the more traditional ones are based on textual form, recent developments in menu-based systems appear to provide new directions in CRL design. Menu-based command languages free

the user from having to direct any effort to conform to a syntax in specifying commands. The developments in using *pop-up menus* now allow the user to apply commands in the visual context of the operands. As well, they provide features like previous command invocation, and command chaining that were possible mainly with the more sophisticated textual command languages.

The CRL for MUPE-2 is aimed to provide the user with a versatile command set and the feature of applying sophisticated combinations and forms of the commands. The command menus of MUPE-2 are designed for better understandability and faster usage. Thus only a small set of options are present for all commands and they are usually available in a single pop-up menu. The versatile commands are driven by the context of the operation, thus providing uniformity in the CRL [MadCR85].

The possibility of using colour in the user interface has also shown new directions in designing the CRL for such environments. In systems like MUPE-2 that are used for development purposes, the 'product' developed has to conform to certain constraints during this process. To facilitate a cooperative effort between the user and the system, the latter normally provides responses to the user, on the consistency of the user's input, in an incremental manner. Thus the major responses of the system to a user's action would be to indicate if the action, the editing operation, resulted in a normal state, an erroneous state, or a state that is open to interpretation, i.e. not fully specified. Such responses in MUPE-2 are indicated with colour in the display. The object of an operation is, in almost all cases, displayed on the screen. Thus by colouring the object in an easily understood colour, it is possible to indicate *quickly* the conflicts with any pre-defined constraints that may have resulted from the user's action. The present scheme uses a predefined set of colours to indicate *errors*, *cautions* and *frozen* operations.

The specification of commands in MUPE-2 follows the model of Star [Smith82]. Most commands take the form of noun-verb pairs. The object of interest (the noun) is first specified and then a command is invoked to manipulate it (the verb). The objects in MUPE-2 are Modula-2 language structures, or constructs, and an object is specified by making a selection. The selected object becomes the 'cursor'. Cursor movements are

possible in a number of ways.

(i) With the pointing device, or the mouse - The tracking object is placed over the object to be selected and the Select mouse button is clicked.

(ii) With the cursor movement keys - There are four types of cursor movements, *in*, *out*, *next* and *previous* that are directly driven from the keyboard. The cursor movements have been designed to provide their response depending on the context. The *in* and *out* movements take the cursor down and up the hierarchy of structures, respectively. The *next* and *previous* movements take the cursor to adjacent structures that allow editing operations. Editing operations not permissible on a particular structure would make the associated commands non-selectable by the user.

The commands of MUPE-2 are generic and perform in the same way regardless of the type of language structure selected. The basic nature of these commands provide for application-independent semantics and result in a command set that is easy to learn and understand. Some example commands are **INSERT**, **REPLACE**, **MOVE** and **DELETE**.

The **INSERT** command is used to insert new structures in the vicinity of the selected structure. The original structure does not undergo any changes. This command is always qualified by an option that specifies whether the insertion is to be before, after, around or inside the current structure. While the meaning of before and after are straightforward, **INSERT** qualified by around means that the new structure should introduce a new level in the hierarchy of structures between the current structure and its parent, if any. **INSERT** qualified by inside introduces the new structure at the level of the children of the current structure. However all the commands and options are not applicable for all structures. The new structure that is to be inserted can be identified either to be a template of Modula-2 structures, or to be replica of an existing structure, provided that the language rules are maintained in the resulting structure.

The **REPLACE** command operates similarly, but it replaces the current structure by a new structure. The **MOVE** command is used for moving existing structures to the vicinity of the current structure. But this command is much more powerful than the

**INSERT** command since it introduces changes at the place where the moved structure is brought from. However it is useful for moving entire program fragments in integration purposes in the building block model of software development. The **DELETE** command deletes the current structure.

# Chapter 3: Architecture of the Editor

The architecture of the system has been designed with the primary goals of portability and adaptability. The present implementation is being targeted to the state-of-the-art graphics based computer workstations [BecBP82, Nicke84]. While these systems provide similar functional capabilities, their application interfaces are far from being standard. The application interface referred to here includes the interface to the graphics library. The UNIX(tm) program development support is the *de facto* standard for these workstations.

As evident from the previous chapter, there is a considerable amount of graphics processing that needs to be carried out frequently, in order to keep the user-interface view consistent with the internal states of the program fragments. Thus, achieving portability in the face of non-standard graphics interface determines the primary dichotomy in the system architecture. The Screen Manager (SM) represents the interface of MUPE-2 with the workstation's intelligent graphics support. It can also be termed as the *Pseudo Workstation Graphics Support* because of its independence from workstation specific details. The support by the Screen Manager could, in principle, also be extended to alphanumeric terminals because of the presence of window managers for such terminals [Engin85]. However, this will not be considered here because of the performance degradation that is anticipated.

The major component of MUPE-2 is the General Manager (GM). In object-oriented terminology it is the 'client' that a user would be associated with. Most of the GM's activities would be performed by requesting services from the diverse 'servers' present in the system. While the system implementation work is not being carried out in a true object-oriented environment, the information hiding concept present in Modula-2 is very 'close' to the object model by its facility of hidden types.

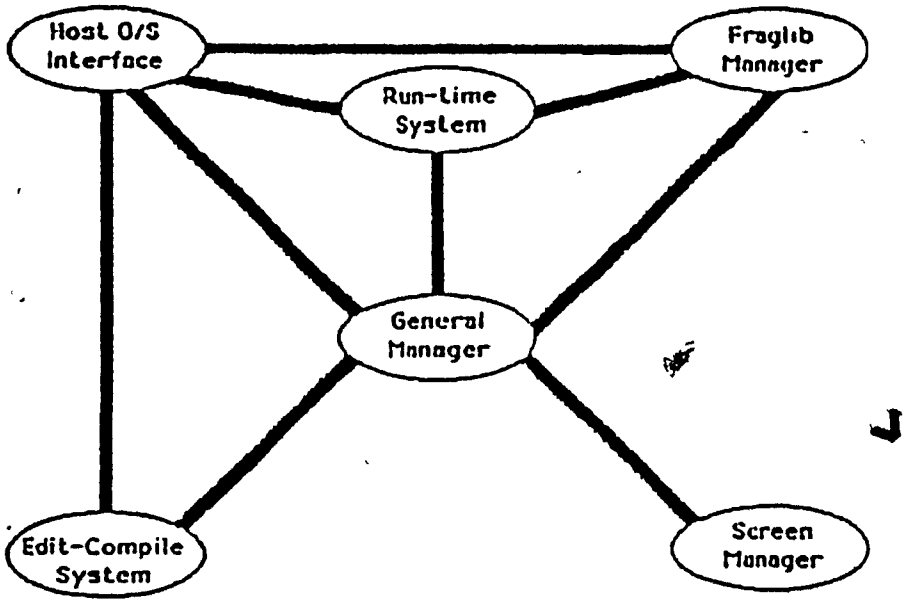


Figure 3.1- The MUPE-2 System Architecture

The GM receives user input from the SM. If the user input is with regard to editing operations, the command or partial operation specification thus received is passed on to the Edit-compile module. In the present implementation the Edit-compile module does not exist as a separate process. Communication with the GM is carried out strictly via the parameters of the procedures of the Edit-compile module that are visible to the GM. The communication is hence synchronous with control returning to the GM only when the Editor completes the execution of the procedure that the former had called. This mode of communication is different from the one between the SM and GM which is asynchronous with a message passing model. The SM-GM communication is examined in detail in a later section.

## 3.1. The Screen Manager

The SM is designed so as to provide the GM with a uniform interface to the display handling mechanism in a workstation independent way. It offers features like:

- Simple Graphics Editor.
- Mouse Position Reporting.
- Keyboard Input Reporting.
- Easy to use pop-up and pull-down menus.
- Specialized Window Manager.

It should be noted that the SM is functionally similar to the Brown Support Environment that has been used in the PECAN system [Reiss84b]. Moreover, similarities can be found between the SM and the *Presentation Component* of the Seeheim UIMS model [PfafH85].

The SM runs as a separate process from the GM. In the present implementation with Modula-2, one of the two is arbitrarily started with a TRANSFER call while the other is started next by a standard procedure call. This takes place in the initialization of the MUPE-2 driver routine. As will be explained in the section on communication, the two managers operate as coroutines and transfer of control takes place each time a message is passed between the two.

The SM provides a top-level drawing of non-overlapping windows and a window hierarchy for overlapping windows, within the non-overlapping windows. The SM sends mouse clicks as the window in which the click occurs and the relative coordinate position within the window where the click took place For pop-up menu selection it sends the item number that was selected. For pull-down menus it sends the command number with which the menu is associated as well as the item that was selected. Keyboard input is handled depending on the nature of the key depressed. For cursor control keys, e.g. the up-arrow and the down-arrow, there is no buffering and the key values are passed to the GM immediately. For the other keys, local buffering is performed and on receiving the string termination character, the string received is transmitted to the GM. Local echoing is done at the current active window to provide the user with the necessary feedback. The textual input would be checked for correctness, as per the language rules, away from the SM. Thus correct textual input would affect the internal representation of some program fragment. The display and the associated screen maps are updated by the

unparsing routines that are called when the internal structures are modified. This may result in text being drawn twice on the screen, once in echo to the input, and the second time the unparsed output. This is unavoidable since the editor allows for entire program fragments to be typed in before validation, and immediate echoing has to be done.

Each message received by the SM from the GM communicates a screen updating. The size of the windows and their relative positions on the screen are however handled by the SM. The GM assumes a standard size for all windows. Activities local to the SM allow the user to alter the window's maximum x and y coordinate values, keeping the corresponding minimum set constant. Note that the effect of this, while similar to scrolling, does not constitute proper scrolling. The difficulty in providing scrolling is in the maintenance of *screen maps*. These maps allow the GM to identify mouse clicks with the program structure(s) associated with that region. Allowing scrolling would entail a communication overhead that would have a degrading effect on the response time.

To summarize, the Screen Manager presents a *workstation-independent* interface for input and output functions for the system. The motivation for such a design was to meet one of the major requirements - portability. The SM is essentially a lower layer on which the rest of the system depends on to communicate with the user. The other parts of the system represent implementations that can be ported easily depending on the availability of the language compiler and the operating system interface.

## 3.2. The General Manager.

The GM has been designed to ease adaptability and extensibility of MUPE-2. While the present thesis is primarily concerned with the design of an editor, it is the GM design that supports the *object-oriented* Editor design. The Edit-compile module is just one resource that the GM could use. The use of such a scheme is helpful for extending or adapting the MUPE-2 system to new and additional features. Thus it is possible to change the Editor's server modules easily if an environment for a different language is required. The modular nature of the design enables changes for the system to be localized if the interface specifications can be maintained.

The editing model that is assumed by the GM is that an operation consists of the following sequence

<operation> ::= <destination> <command> [<option>] [<source>]

The destination of the operation is always at the current structure of interest as indicated by the active cursor. The tokens for the above are identified with the help of the screen maps available to the GM. Once identified, the tokens are sent to the Edit-compile module. Basically, the GM does not check for operation specification completeness. This was necessary in the design because of MUPE-2's feature of user interruptible operations. It essentially allows the user to have an incompletely specified operation and initiate the specification of another operation. The incompletely specified operation is then interrupted and can be resumed at a later instant. This feature is specific to the Edit-compile module and hence it is handled there. The mechanism for handling interrupts and the features available will not be discussed further in this thesis, but can be found in [MadhP85].

In the initialization of the GM process, performed at MUPE-2 start-up, the commands that are active at this time are enabled and the initial screen is drawn. The available command set is always made known to the user via the display. The availability does not remain constant over time because it depends on the context of the user's operation, i.e. the active cursor.

At initialization the drawing of the entire screen is performed by the GM. The drawing is achieved by calls to the graphics editor primitives available in the SM. The calls are implemented by a message passing scheme and the calls are non-blocking. The detailed design of the communication is provided in a later section. There would be a considerable communication overhead in this initialization, but because the full screen need be drawn from scratch only once, the payoffs in using message passing are still attractive. What the message passing scheme tries to achieve is to enable the two processes to work independently with a minimum of coupling. With such a scheme the extensions to a distributed environment will be done in a natural way.

In using the Modula-2 PROCESSES facility to achieve process independence to some degree, it is necessary to devise a procedure which would perform the functions described above. The GM procedure (process) body consists of a loop as below.

```
LOOP
  GetSMMessage();
  AnalyzeAndAct()
END
```

**GetSMMessage** fetches a message from the buffer (or mailbox) associated with the SM to GM communication. It blocks if the buffer is empty. **AnalyzeAndAct** analyzes the message received from the SM for the event reported by the SM. Essentially the SM provides lexical tokens to the GM as the input from user to the system (and it is also true that the SM receives lexical tokens from the system to communicate to the user). The tokens received from the SM are then handled by the GM at the syntactic level to build the structure of the operation. Once the operation structure is complete, editor-semantic routines are called to perform the operation. Some of the token types and the corresponding GM actions are as follows:

(a) Keyboard_Event

(i) Cursor_Control_Key

Update value of the structured cursor so that it points to the new structure.

(ii) String

If there is an incomplete operation at the current cursor position awaiting the specification of a <source>, parse the input string, check for operand compatibility and if compatible, perform the command with the parsed structure at the destination structure. Otherwise do not accept input.

(b) Screen_Item_Selection

(i) Menu_Item

Derive the partial specification for the operation.

(ii) Window_Item

Derive the associated structure from the screen maps. If no incomplete operation specification at current cursor, then update structured cursor, else check operand compatibility for the operation.

The text and graphics displayed within the windows are essentially (partial) views of the structures operated upon by the Editor. These views are obtained by unparsing the internal representation of the program fragments. The unparsing is an incremental process. It has been apparent for sometime [Fritz84] that non-incremental schemes for unparsing are too slow when the fragment being edited exceeds five pages. The more recent schemes, as in $R^n$ [CapHo86], adopt forward and backward incremental unparsing. In MUPE-2's editor, unparsing is triggered either by a modification in the internal tree structure, or by a cursor movement which takes the focus of attention, the cursor, to a structure that is not currently displayed. The incremental unparsing algorithm displays the cursor in the centre of the window and unparses in both directions of the cursor, till each of the screen halves are filled up.

During the unparsing process, screen maps are constructed to map between internal structures and window relative coordinates. When a user clicks at some element, textual or graphical, within a window, the screen maps are accessed to provide the structure identification from relative window coordinates. However, the screen maps are dynamic since the window contents continually change.

While the internal representation of program fragments is out of the scope of this thesis, it is appropriate to consider them in the context of the implementation of the screen maps. The MUPE-2 system maintains program fragments in a variation of the abstract syntax tree representation. Abstract syntax trees have proved to be popular with many implementations because they provide a representation somewhere in between concrete syntax and actual semantics. The unparser generates the concrete representation of a syntax tree from its internal representation. The concrete representation is communicated to the SM for display. The screen map is generated (updated) during the unparsing process. The unparser is aware of the location within the window where the display takes place with each call to the SM display routines.

In systems of the type of MUPE-2, the text that is displayed is often derived dynamically from structured internal representations. Thus to allow the user to 'pick' internal structures from their textual representations underscores the need for

maintaining mappings between screen coordinates and internal structures. This 'picking' facility is available in systems like IPSEN [Nagl83] and PECAN among others. The general method for identifying the internal structures has been to use a table with entries of the pair (Screen Coordinates, Pointer to Internal Structures). However such a method is space consuming and the structure resolution process is complicated. The scheme that is used in the present implementation eliminates a major part of the space consumption with limited effect on the performance.

This method saves on the space requirements because it does not use a table to identify the structures. Since in the display of a program fragment, the current cursor is always in the associated window, any structure 'picked' must be in the neighbourhood of the current cursor. Instead of using a table, the nodes of the internal representation can be *attributed* with the coordinates of the rectangular region within which the subtree below the node is displayed. The attribution of the structure nodes is done at all times of the unparser traversals.

Figure 3.2 is an illustration of a contrived example of a portion of a program fragment with its display and internal representation. The scheme to identify a selected structure is now explained. Let the cursor be the procedure call within the sequence of statements as shown in the figure. Let the mouse click occur at the position indicated by #. The rectangles associated with each of the nodes is as indicated by the dashed lines in the figure.

The search for the associated structure starts at the node numbered 3, and whose rectangle limits indicate that the click did not take place within the subtree rooted at this node. Next the search continues by following the sibling pointer to node 4, then to 1, stopping with success at 2. To obtain finer resolution the search continues to the children nodes of 2. It continues in this manner to finally return the Finegrain structure, which is the appropriate structure.

The searching algorithm in pseudo-code is as follows.

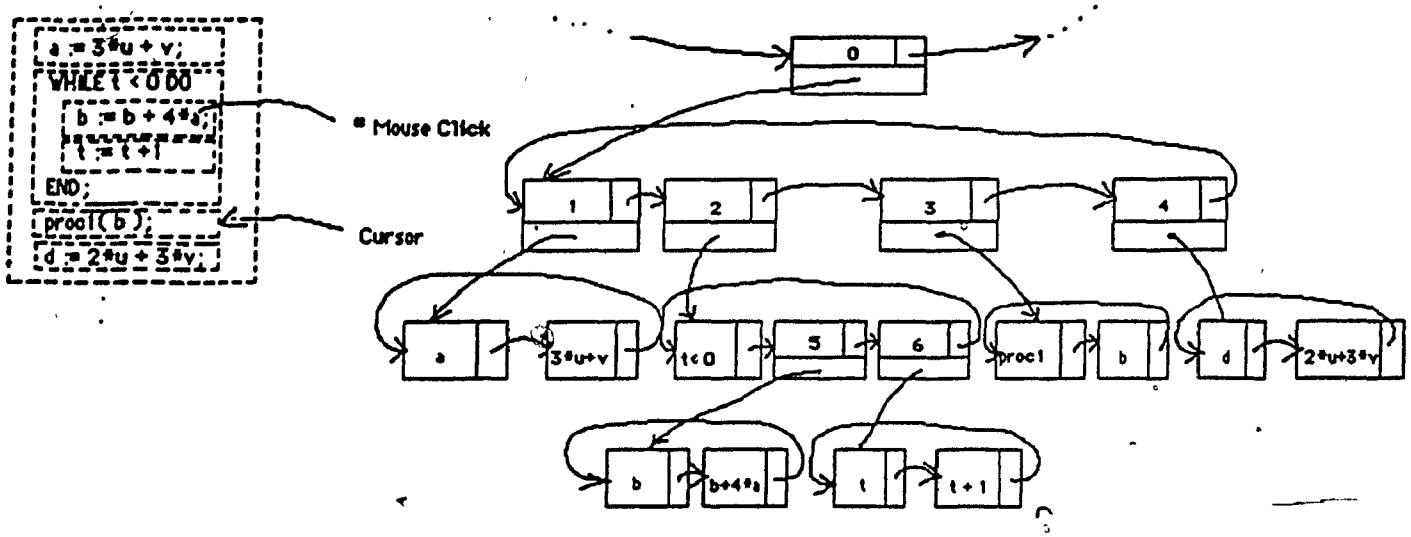


Figure 3.2- Structure Selection in a Program Fragment.

```
PROCEDURE SearchStructure(CurrentPointer:NodePointer;Click:Coordinates):NodePointer;
VAR
  SearchPointer: NodePointer;
BEGIN
  SearchPointer := CurrentPointer;
  LOOP
    IF (Click in SearchPointer's region) THEN
      RETURN ObtainStructure(SearchPointer,Click)
    ELSE
      SearchPointer := SearchPointer^.Parent
    ENDIF
  ENDLOOP
END

PROCEDURE ObtainStructure(PS:NodePointer;Click:Coordinates):NodePointer;
VAR
  S1Ptr,S2Ptr: NodePointer;
BEGIN
  S1Ptr := PS^.FirstChild;
  S2Ptr := S1Ptr;
  REPEAT
    IF (Click in S1Ptr's region) THEN
```

```
    RETURN ObtainStructure(S1Ptr,Click)
   ELSE
    S1Ptr := S1Ptr^.NextSibling
   ENDIF
  UNTIL S1Ptr = S2Ptr;
  RETURN PS
END
```

The loop in SearchStructure has to halt since the reported click is always within the root's region. The sibling nodes form a circular list structure and the search at a level ends when the circle is completed. There is a small amount of redundant work done in examining a node again after the search has climbed to the parent. This can be avoided at the cost of a 'marking' bit.

### 3.3. Single Processor GM-SM Communication

The GM and SM are two cooperating sequential processes. The synchronization and communication of such processes are of primary importance for effective utilization of available resources. Among the many approaches proposed to achieve synchronization and communication among such processes, the popular ones have been monitors [Hanse75] and message passing [Hoare78, Gentl81]. The resulting process structuring in programming via these two approaches are distinct in nature and is one of the criteria used for selecting one over another. An advantage of using message passing is that it is easily related to distributed systems implementation. The process structuring from the *object-message* viewpoint is easy and natural, and the resulting structure translates directly into code.

MUPE-2's strong emphasis on adaptability in the design requires it to be adaptable to the recent trends in distributed operating systems. The design philosophy of these systems is that a workstation should be more than an intelligent terminal. Consequently, the workstation should have a larger role by offering its resources to the overall system. The state-of-the-art workstations in the distributed systems are of the power of the SUN and IRIS variety. The generic features of such distributed systems are presented in the next section where extensions to the present MUPE-2 single processor implementation

would be examined.

The single processor implementation for the SM-GM model is based on the support provided by the SYSTEM module in the standard Modula-2 library. The SYSTEM module is an encapsulation of the low-level facilities that are provided by the particular compiler implementation. However, the SYSTEM module facilities are for single processor implementations alone and do not provide the abstraction necessary for adapting to multiprocessor configurations. Formulation of processes and their interaction, via Modula-2 can be achieved at a higher level of abstraction which hides the underlying system configuration. In a single processor configuration, there is only one process that is executing at any instant and the processor is time-multiplexed between the processes.

To achieve this higher level of abstraction, [Wirth85] has introduced a module called PROCESSES This module provides a small and simple facility for dealing with concurrent, cooperating 'threads of control' (processes). The module PROCESSES provides the following.

1. Procedure StartProcess(P:PROC; n:CARDINAL)

   Starts a process P with worksize of size n.

2. Procedure Send(VAR S:signal)

   Reactivates a process waiting for signal S.

3. Procedure Wait(VAR S:signal)

   Wait for some process to send signal S.

4. Procedure Awaited(S:signal)

   Returns true if any processes are waiting on signal S.

5. Procedure Init(VAR S:signal)

   Initializes the signal S.

6. Type SIGNAL

   Opaque type SIGNAL.

The call to procedure StartProcess is to start the execution of a process expressed by the parameterless procedure P in its argument list. The distinction between genuine concurrency and quasi-concurrency in the execution of P and the calling body is

dependent on the implementation of the PROCESSES module. The communication between processes for purposes of conveying data among one another is achieved by common, shared variables. Apart from such communication, processes also need to interact in such a way as to force a particular sequence on their executions - that is to synchronize. Such communication for purposes of synchronization is handled by *signals*. The data type SIGNAL exported from the PROCESSES module carry no data as such. Every signal is used to denote some condition in the program's variables. Sending the signal implies that such a condition has taken place. For processes waking up to a signal, the subsequent operations are based on the assumptions that the condition has been met

The implementation of the PROCESSES module for single processor Modula-2 compilers is based on the previously mentioned SYSTEM module, which provides the *low-level facilities* that enable Modula-2 programs to consider machine-dependent specifications. Central to the implementation of the PROCESSES module via the SYSTEM module is the notion of *co-routines*. Coroutines are sequential programs that can be executed quasi-concurrently. In such a situation, the processor is switched from one coroutine to another by explicit transfer statements. The SYSTEM module provides a TRANSFER procedure to achieve exactly this.

In the implementation of the PROCESSES module provided in [Wirth85], each process started by a **StartProcess** call is represented by a process descriptor in a process ring. The ring contains descriptions of all processes created up to that time. Processes operating on the same signal are further threaded together within this ring. **Send(S)** takes the first element off the thread associated with S and transfers control to it from the calling process. **Wait(S)** places the calling process at the end of the S thread and control is passed to the next ready to run process from the ring. Thus a fair queuing policy is implemented by the use of the threads.

For the implementation to reflect a message passing scheme in process communication, the *monitor* approach for mutual exclusion has been followed. Messages are the shared variables (buffers) whose access is protected by the synchronization facilities provided by PROCESSES. The message passing scheme designed is fairly simple because of

the presence of only two processes, the GM and SM. The *objects* in this message passing scheme are quasi-concurrent processes that deposit messages in the named buffers to communicate with other objects. When an object is *free* it fetches messages from the buffer associated with it. Since there are only two such objects, the identification of the sender of the message is not necessary to be part of the message.

The buffers are finite in size. Any attempt to deposit messages into a full buffer or to fetch from an empty buffer results in the process being put to 'sleep' with a Wait for the appropriate signal. It is thus an implementation of the well-known Producer-Consumer problem. However, we have decided to approach the problem from the message-passing point of view because it offers us a methodology for adapting to (concurrent) distributed systems. The duality between monitors and message-passing for communication and synchronization has been apparent for some time [LaueN79].

The PROCESSES module, while adequate for the present implementation, has some limitations that precludes it from general acceptance. A major difficulty with Wirth's PROCESSES implementation, is its awkward semantics when a process or the main program terminates. As soon as such a termination takes place, the entire program terminates. Thus it does not support a dynamic multiprocessing environment where processes are regularly initiated and terminated. This problem has also been addressed in [Sewry84], where in a modified implementation of PROCESSES, all processes call a standard procedure as the last statement. This procedure would remove the corresponding Process Descriptor from the schedule ring maintained by PROCESSES, and transfer the control to the next ready to run process from the ring. While this is a major improvement in concurrency control via Modula-2, it does not unduly influence the modeling of SM and GM as processes. For our implementation, SM and GM are static processes and the two do not exist separately. The SM could not exist 'meaningfully' without the GM because the only intelligence that the SM has is in the area of 'terminal' management. The GM's communication with the user is handled by the SM. Without SM, GM is isolated from any interaction with the user. The final termination is brought about by the user communicating a *quit* signal to the GM which takes it out of the main loop in its body and on to the procedure end and hence to the end of the entire program.

### 3.4. Extensions to a Distributed System

With the growing popularity of powerful workstations and high speed local area networks, there has been a lot of research on the structure of applications in distributed systems. In many of the present workstation applications, the workstation is either treated as a remote terminal for a powerful mainframe, or it is treated as a stand-alone personal computer. However, the present day workstations are better utilized if they are to be treated as a multifunction **component** of a distributed system. The reason for this is that the power of the workstation will not be wasted in treating it merely as a terminal. Furthermore, using it standalone isolates it from the benefits of the computation power available in the larger range of computers

One of the most important functions for the workstation in such a distributed system is in providing the user interface support. The workstation acts as a front-end for the resources, which could be local or remote. Tasks down-loaded from a mainframe frees it to concentrate on the computation intensive tasks it is best at

For a distributed implementation of MUPE-2, it is intended to use the workstation as such a component. The present design is adaptable to such a configuration because of the layered approach that has been adopted in the design of MUPE-2. The model of distributed computing that attention is focussed on is based on the V-System [LantN84] The V-System is an environment consisting of workstations, standard time-sharing systems and dedicated server machines, interconnected by various local area networks including the Ethernet [MetcB76]. It is representative of the recent workstation-based distributed systems. The fundamental software architecture is that the system is functionally decomposed into modules such as workstation agents or managers A module could act as a server of a particular resource, or as a client for some other resource, or both. The clients and servers may be distributed throughout the network with the same semantics for local or remote access, or communication.

In such an implementation of MUPE-2, the SM is the 'terminal' server. Futhermore, the SM is designed in an application independent manner and can be used for applications other than the MUPE-2 system. The main changes that are foreseen in a

transition to a distributed implementation would be in process scheduling, and process communication. The syntax and semantics of message passing have been designed so that they can remain virtually unchanged. The structure and capabilities of the SM would undergo minor modifications with its adapting some role in communicating with a distributed kernel in the identification of remote servers. The present system architecture has been designed so that the partitioning of tasks is the appropriate one to achieve the best performance in the distributed setting.

In the distributed implementation, the SM would be the only component that would be present in the local workstation to the user. The GM and the Editor modules would exist in a remote and more powerful computing system. The SM design is based on the Virtual Graphics Terminal Service (VGTS) that is present in the V-System. VGTS supports a wide variety of structured graphics in an application and device independent way

Thus in conclusion, this section has presented the case for distributed systems based on the power of present day workstations. The structure of a similar recent system, the V-System, was examined to understand and appreciate the performance of such systems. The adaptability requirement of MUPE-2 guided the design of the SM-GM communication. The transition of MUPE-2 from a centralized implementation to a distributed one should be a simple one. Consideration of SM-GM communication in such a light would provide major benefits at such transition time.

# Chapter 4: A Scheme for Enforcing Syntax Rules.

A major part of the design of a language based editor is concerned with incorporating the context-free syntax rules of the language. Being context-free, these rules signify conditions that are to be enforced in localized language constructs. In a conventional compiler, these rules are the basis on which the parser is constructed. However, in a template driven language based editor environment, the front end of a conventional compiler, i.e. the parser, is not necessary. Here the role of the parser is played by the mechanism that decides the set of templates and the structure types that are available for a particular operation on a particular structure. This set is selected to maintain the program in a consistent state by the context-free rules of the language. Whether anything else takes the program to an inconsistent state is the subject of the next chapter.

This chapter aims to present a scheme to enforce the language's syntactic rules in a template driven language based editor. An introduction is provided to a taxonomy in language based editors, and the model of the MUPE-2 editor is presented within such a classification. The primary concerns in developing the scheme have been that the implementation should prove to be adaptable and portable. Attempts to meet the adaptability goal have been made by utilizing a table-driven approach in the specification of the language rules. However to design an efficient table organization requires a classification model of language structures on which the editor operates, and doing so, to develop a general model for editor operations. Such a model is derived in this chapter. Portability in the editor's implementation across a wide variety of issues is essential. As an example, this editor has been designed to be portable across the internal representation of the program structures. This is achieved by the use of a structured value to represent the 'cursor' in the editing operations. Thus the structured value of the cursor provides an abstraction of the program structures that the editor operates on. The tradeoffs in making these implementation choices are finally noted.

## 4.1. Types of Language Based Editors

Language based editors are usually of two types. In the first type there is an incremental parser which is invoked at various points during the course of editing. Thus, as a program is being built with such an editor, parsing is carried out (perhaps for each line that is typed in) to maintain the syntactic correctness of the program. The incremental parser is normally able to notify the user immediately that the (previous) input was incorrect. However, there are some editors, such as the one in COPE [ArchC81], which would perform error repair on the input such that no error messages need to be indicated. This is part of a philosophy of cooperative programming between the system and the user. There could exist on-line help facilities to provide the user with the syntaxes that are acceptable at any point within the developing program.

The other type of editors to incorporate language rules are the template driven editors. These editors are usually associated with a menu based command language for the editor. In such systems, the user is provided with a choice of language constructs that are appropriate for replacing a non-terminal of the language. Here once such a construct is chosen, its expansion is included in the program being developed. Thus the user is saved from the bother of having to remember the correct spelling of keywords, or to match *begin-ends*, among others.

The process of such a selection is equivalent to applying a particular production of the grammar at a non-terminal. As a production of the grammar may have non-terminals on its right hand side, the language constructs could have portions that need further expansion Thus the language constructs chosen could appear as templates with placeholders for the portions that are not fully expanded. For the scanner, the only requirement could be to identify identifiers and constants. However, a few editors, like the CPS [TeiRe81], use an expression parser which could be seen as freeing the user from the lengthy process of inserting an expression by the step by step expansion of *term*, *factors* and others that are included in the language rules to enforce operator precedence.

In the MUPE-2 editor, textual input and templates are used through all levels of the grammar. This is similar to the approach taken by editors like SYNED [Gasne83] and

PECAN [Reiss84a] in offering *bimodality*. The main reason for this is that the editor should be convenient to users with a wide spectrum of familiarity with the language. While the template driven mode provides a strict adherence to the language rules, users fairly familiar with the language normally find it restrictive. For such persons the template driven mode is useful to construct the overall structure of the program, but would prefer textual input for the rest of the program body. To provide the textual input facility at all points in the program require constructing a novel parser of the type described in [Wegma80], which uses a set of heuristics in its parsing. This thesis will not deal further with such a parser since it is still under design. The editor is currently being developed with only a parser for elementary structures in the Finegrain domain Such structures will be introduced in a later section.

## 4.2. Editing Operations

In the present editor, templates are chosen as part of the operation specification. The specification starts with the selection of a command like **INSERT** or **REPLACE**, and the specification completes, in most cases, when the source operand for the operation is specified. Once the operation is fully specified, the 'action routines' to carry out the operations are triggered. These commands are implemented by combining the primitive tree operations of grafting and pruning. As described in an earlier chapter, the target of these operations is always at the location of the current cursor.

The following is an example illustrating the specification of an operation.

Consider the current cursor to be located at the **while** statement in the diagram shown. The user selects the command **INSERT**. This command has to be further qualified by an option from a subset of {**Before, After, InsideFirst, InsideLast, AroundFirst, AroundLast**}. The options available are presented to the user in a menu of options. The available options depend on the structure that the cursor is on, and the command that has been selected. In the above selection of the insert operation, the available option set is the full set as shown above.

<designator> := <expression>;

```
WHILE i >= 0 DO
  Subtotal[i] := Credit[i] - Debit[i];
  DEC(i)
END;
```

A short explanation on the significance of the options for the insert operation follows. InsideFirst and InsideLast distinguishes whether the insertion is to the first statement or the last statement of the while loop in the example. On the other hand, AroundFirst and AroundLast are meant to distinguish in the insertion of an existing structure around the target source structure. Since such an existing structure has to be of a 'container' type, AroundFirst places the destination structure as the first structure in the container. Similarly, AroundLast makes it to be the last.

The necessary option sets for a command is derived from the cursor information and the command. In this editor a table representation has been used to store the appropriate option sets. The prime motivation for the use of table representation is that it allows for flexibity and adaptability in the editor. Modification in a table entry is, in general, a simpler process than modification of logic incorporated as code in a program.

The design of the tables for accessing the option sets is decided by the mechanism used for incorporating the cursor information. However cursor information has a larger role to play in the further specification of the operations. Hence description of the tables for accessing option sets will be postponed until the operation specification description is completed.

In typical operations involving tree structures, portions of the tree are replaced by an already existing subtree (by copy or transfer), or by the application of a production from the language rules. This requires a further step in the specification of the operation. In this step the source of the operation is specified. As already described, the entire operation could be seen as:

<operation> ::= <destination><command>[<option>][<source>]

For operations that do not have a source operand, the specification is in postfix form. However such a specification method applies only to one operand operations. For operations with source as the second operand, the specification is in infix form. This is in contrast to a specification scheme which would have allowed 'selection' of a number of structures and then applying the single command on the selected structures Such schemes are present in many of the recent graphics-based editors, but are not appropriate when language rules have to be obeyed, which essentially is to maintain a larger overall structural rigor.

The source can be specified either as an existing program structure, as textual input which is typed in at that point, or as a language template. In essence they both imply the application of a production of the language's underlying grammar. The source being an existing program structure implies the application of a production of the grammar where some of the right hand side nonterminals could be already expanded. A language template is necessarily a production where none of its nonterminals are expanded.

Since protection is a major requirement for the editor, the *candidacy* of a program structure in an operation needs to be first validated before the routine implementing the operation is called with the structure chosen as the source. The scheme used for this validation depends upon a hierarchical classification of the program structures. Using such a scheme enables the validation routine to make quick decisions since a coarse filter is sufficient to discard many of the non-permissible program structures.

Language structures that are identifiable in any given segment of a program can be seen to belong to classes that are easily distinguishable from each other. Thus a language structure like an assignment statement is easily distinguished from one that is a constant

declaration, which in turn is again different from an expression. Elements of one such class can, in general, be seen to be **composed** of elements of other classes, and elements of a particular class when ordered with elements of another class can **form** elements of a third class. Thus the editing operations that a software development environment like MUPE-2 provides, are essentially going by rules that maintain these class distinctions. Moreover, these editing operations truly reflect a bottom-up, top-down, or 'same-level' nature of software development. As a result, the editing operation rules specify a *mapping* from a set of classes to another set of classes. Since in an editing operation, there are the destination and source operands, and a resulting structure, a *signature* of a generalized editing operation $EditOp_x$ is as,

$$EditOp_x \cdot Class_i \; X \; Class_j \rightarrow Class_k$$

This means that the editing operation $EditOp_x$ is applicable to structures as are specified above in the signature. $Class_i$ is the class of the destination operand, $Class_j$ is the class of the source operand, and $Class_k$ is the class of the resulting structure.

The interest then is to be able to formulate the editing operations by such mappings, and thus to drive the editing machinery. Thus for every editing operation, a set of tuples or a *relation*, defines the rules for applying the operation. To achieve this, the classification scheme of language structures is the topic of the next section.

### 4.3. Structure Classification

The invariant of an editing operation is that, at the syntactic level, the resulting structure is still a sentence of the context-free grammar rules describing the syntax of the language. However these grammar rules are presented in such a way that to derive the rules for editing operations requires an initial analysis phase. As an example, consider the editing operation **INSERT-Around** on a structure which is a **statement** of the language. It is possible to use a **procedure** as the *source* for this operation if the statement is the only language structure of the fragment. In any other situations such an operation would not be possible. Thus the language structure classification scheme that should be adopted to denote the allowable set of editing operations must make such

distinctions. Furthermore, many of the editing operations apply for a wide range of classes. For example, the **INSERT-After** operation applies for all classes of statements whenever the *destination* operand is of any statement type.

Thus, the classification scheme that is most suitable for this purpose is a hierarchical one. This is so because it is then possible to incorporate the rules of such language dependent editing as compatibility in the different levels in the hierarchy. This results in situations where the satisfiability requirements are limited to examining the classification values in the higher levels of the hierarchy This introduces the aspect of *precision* in structure specification. In other words, the editor does not require the same amount of information of the language structures, when it derives the validity of language based editing operations.

The classification model that is used in the editor is presented in this section in a formal notational basis. The application of this model to some actual Modula-2 structures is also illustrated in this section.

A program fragment (F) is composed of (→) a sequence of well-formed program structures or language constructs ($S_i$'s).

$$F \rightarrow S_1\ S_2\ S_3\ \ldots\ S_m$$

Each of the program structures ($S_i$'s) can either be a compound structure, in which case it is composed of inner program structures, or it can be an atomic or **Finegrain** structure. Thus for a compound structure $S_k$,

$$S_k \rightarrow S_x\ S_y\ \ldots\ S_z .$$

The classification of a structure derives a structured value for that structure as

$$\text{class}(S_i) = (D_{1k}, D_{2l}, \ldots, E_z).$$

This classification of a structure exhibits the hierarchical classification value. That is, among all possible structures in a classification tree, the structured value of class ($S_i$) denotes the path from the root of the classification tree to its leaf node, $E_z$. Thus for the structure $S_i$, $D_{1k}$ above denotes the top-level domain value, the subsequent elements the

sub-domain values, and finally $E_s$ the actual structure (generally a pointer to an internal representation, or 'opaque' type). Note that this classification is independent of the internal representation that is used to store the structures. The function **class(.)** is available at the interface of the internal representations to provide the classification value. Thus the internal representation could be of any structural form like abstract syntax trees, directed acyclic graphs, or parse trees.

The top-most classification of language structures is by the primary domains. Primary domains identify the *granularity* of a language structure. While the Fragtypes of MUPE-2, as introduced in chapter 2, are based on such a granularity concept, there are domains to classify inner structures that are not allowed to exist as a fragment of their own. These structures are not *well-formed* in the sense that it is not possible to identify them as building blocks, but are more appropriate in considering the user interface of the editor vis a vis cursor movement. While the concepts behind, and the facilities of, *cursor movement* will not be dealt with in the thesis, considerable attention has been paid to the ease of cursor movements in MUPE-2's structure editor. In the following the primary domain categories in structure classification and the possible subdomain constituents within such domains are discussed.

Independent to the primary domain value of a structure, every structure also holds contextual information as to whether it is solitary and whether it is grouped. A true value for solitary applies only when the cursor structure is the only structure in the fragment. A grouped structure indicates a pseudo-structure obtained by **GROUP**ing adjacent structures. The primary domains are Finegrain, Statements, Partial Statements, Declaration, Types, Partial Types, Record Fields, Nodes, and Fragment Level.

Finegrain structures are essentially textual structures and are designed to provide uniformity among the fine granularity structures of the language like expression, identifier lists and parameter lists. The textual mode of editing is the only way such structures may be modified. The role of determining whether the text obeys the language rules in the context it appears, is played by the Finegrain parser. Such a parser determines the acceptable nonterminals from the internal representation where the text has

been input, and parses the structure based on the collected information.

Structures with Statements as their primary domain are the statements-level structures of Modula-2. The structures within such a domain are further classified as belonging to one of Containers, Composite Containers, or Textual subdomains. Belonging to the Containers subdomain are the looping constructs and the WITH statements. Composite Containers apply to the IF and CASE statements. These statements need to be distinguished from the previous type because the nature of their cells differ from those of Containers. An IF statement contains a THEN-part, possibly more than one ELSIF-parts, and possibly one ELSE-part. Similary, a CASE statement contains one or more CASE-elements. The cells of the IF statements are of identical classification and the distinction among the different 'parts' is made by the unparsing routine (similar to 'guards'). Statements in the Textual subdomain do not possess a structural composition and thus prohibit operations with **Inside** as an option. The assignment statement, the RETURN statement, and the EXIT statement belong to this subdomain.

Partial Statements subdomain designates the contents of IF and CASE statements. They are structurally composed of an expression and a single body of statements. There is no further classification among these structures for the purposes of syntactic distinction. An example illustrating Partial Statements is the boxed structure shown below.

```
IF ch >= " " THEN
    x := ch;
    EXIT
```

```
ELSIF ch = LF THEN
    x := " ";
    eoln := TRUE;
    EXIT
```

```
ELSIF ch = FS THEN
    x := " ";
    eoln := TRUE;
    eof := TRUE;
    EXIT
END
```

The Declarations domain designates structures that represent declarations of constants, types and variables in Modula-2. Procedure and Module declarations are not included in this domain because the graphical interface allowed displaying the static scoping tree, and editor operations in such a view and hence need to be treated differently. Procedures and Modules are included in the Nodes domain.

The domain of Types is an example of a domain covering structures that are not permitted to exist independently as a fragment. However such structures can replace existing or unrefined type definitions. These appear on the right-hand side of type and variable declarations. An example structure is shown boxed below.

```
VAR
    a : ARRAY [0..N-1] OF CARDINAL;
```

Substructures of structured Types structures as shown above belong to the domain of Partial Types. These too are strictly inner structures that do not exist as independent fragments. Operations on these structures permit modifications to the structural class of a Type without changing the base type. The boxed structure below is an example of such structure.

```
VAR
```

a : ARRAY [0..N-1] OF CARDINAL;

Record declarations in Modula-2 involve structures within it that are different from those found in other type declarations. Thus the domain of Record Fields cover the field structures to be found in a record declaration. The full classification of the structures in this domain requires distinction between variant and non- variant fields, between ELSE fields and non-ELSE fields, among others. The boxed structure shown below is an example.

```
Person = RECORD
    lastname, firstname . Name;

    CASE male : BOOLEAN OF
      TRUE : MilitaryRank : CARDINAL |
      FALSE : MaidenName : Name
    END

END;
```

The Nodes domain cover Procedure and Module declarations either individually, with their children Procedure and Modules, or when GROUPed with sibling declarations. In addition, MUPE-2 environment rules permit declarations of UNITs, which are unrefined structures that can subsequently be refined to either a Procedure or a Module declaration. Nodes with their children are referred to as Subsystems, and collections of sibling nodes are referred to as System-layers. The classification within this domain includes differentiating between internal nodes and root nodes, between 'compilation-unit' nodes and others, between the screen on which the editing is being performed, among others.

Fragments are the entities that are provided with independent existence in MUPE-2. However, editing operations are permitted on Fragments in the same way as structures

within a Fragment. But there is difference between a Fragment as a whole and the contents of Fragment. Thus it is possible to delete the entire contents of a Fragment and result with an empty fragment. Deleting a Fragment however removes all traces of the Fragment from the environment. Furthermore, there are Fragments in the subdomain Abstracts, that do not obey the language rules on which the environment is based. These have been designed mainly for documentation purposes.

The following is the classification tree for structures applicable for editing, expressed in EBNF notation.

| | | |
|---|---|---|
| Cursor | ::= | (Solitary\|Nonsolitary)(Grouped\|Nongrouped) Domain |
| Domain | ::= | Finegrain \|Types\|PartialTypes\|RecordFlds\|Declarations\| Statements\|PartialStmts\|Nodes\|Fragments |
| Finegrain | ::= | ContainerFinegrain\|NoncontainerFinegrain |
| Types | ::= | Textual\|Structured |
| Textual | ::= | ListStructured\|NonListStructured |
| Structured | ::= | ProcedureType\|RecordType\|ArrayType\|PointerType\|SetType |
| PartialTypes | ::= | PointerToText\|PointerToStruc\|SetOfText\| ArrayOfStruct\|ArrayOfText |
| RecordFlds | ::= | (VariantFld\|NonVariantFld)(LastFld\|NonLastFld) (CasePart\|NonCasePart)(ElseFld\|NonElseFld) |
| Declarations | ::= | ProcedureType\|NonProcedureType |
| Statements | ::= | (Container\|StructuredStmt\|TextualStmt)(IfStmt\|NonIfStmt) |
| PartialStmts | ::= | (CaseElem\|IfElem)(LastOne\|NonLastOne)(ElsePart\|NonElsePart) |
| Nodes | ::= | (SubtreeRoot\|Subtree\|ChildlessRoot\|TheRoot)(CompUnit\|NonCompUnit) (UniqueNonUnit\|NonUniqNonUnit)(ScratchPad\|ModScreen\|ProcScreen) |
| CompUnit | ::= | (ProgModule\|DefImpModule\|DefMod\|ImpMod\|Unit) |
| NonCompUnit | ::= | (InternalNode\|NonInternalNode) |
| Fragments | ::= | (Expressions\|SingleStmt\|Stmts\|SingleDecln\|Abstracts\|ProcModule\| ProgDefs\|CompUnit\|NonCompUnit\|UnitSubSystem\|UnitLayer\| CompUnitLayer\|NonCompUnitLayer) |

The classification information for a structure is a hierarchy of information and it allows a range of precision in specification. However to maintain such a classification information at every structure would greatly increase the storage requirements for fragments. This can be avoided if the structure classification can be *demand-driven*. This means that the classification information for a structure is evaluated only when the compatibility of operands (structures) for an operation is to be determined.

The process of classifying structures is carried out in at most two separate instances during an editing operation. At any time during editing a fragment there is a *current* structure of interest. Any editing operation that is performed, always takes the current structure as the destination. Furthermore, the available editing commands and their options, if any, are derived from the attributes of the current structure. Thus the current structure needs to be fully specified at all times. The specification of the current structure forms the structured value known as the *cursor information*. This is the first instance of structure classification, and it is performed any time that the current structure of interest changes. The other instance of structure classification is when an editing operation is initiated and the source for the operation is indicated as a structure. Then to check whether the indicated structure is compatible for the command specified on the current structure, the structure is classified accordingly.

## 4.4. Table-driven Menu Generation and Compatibility Checking

Having illustrated the cursor classification scheme, this section presents the table driven implementation for the display of menu items. Another aspect of the editor that will be examined in this section is the design of an efficient scheme for compatibility checking in editing operations.

Menus appear at two instances in the specification of the operation Firstly, menus are presented for commands that need to be further specified by an option. An example of this is the **INSERT** command. It is to be further qualified by one of the options that are available for the application of **INSERT** at the structure where the cursor is currently positioned. The decisions involved in making the appropriate display of menu items are done using the current value of the structured cursor. It should be noted that the structured cursor is determined or evaluated whenever it is moved to a new structure. Thus when it comes to displaying menu items no time need be spent in first determining the cursor value. This is critical since fast response is necessary when displaying *pull-down* menus. While such a scheme may slow down the response in cursor motion the trade-off is justified.

There are only two commands, **INSERT** and **MOVE**, where option menus need to be shown. Thus it is not feasible in this case to have Command as a table dimension since a single conditional statement is sufficiently quick to make the decision. In addition, these two commands have no differences in option set values at all cursor positions.

The second instance where menus are used during operation specification is in operations that allow language templates as the 'source' in an operation. As indicated in an earlier section, specifying a language template to be installed at a particular position in the program tree is equivalent to the application of the corresponding production of the language. In the present editor the commands that permit template specification are **INSERT** and **REPLACE**. The templates that are available for a particular operation are decided by the combination of the command, the option(if applicable) and the cursor. It should be noted, however, that the user is allowed to choose an item from the menu of templates, or select an existing program structure as the *candidate* for the operation. Thus the user actions undergo the following sequence:

(1) select a command from the commands that are available at that context,

(2) if the command requires further specification of an option, select such an option from the menu that is presented, and

(3) if the command selected was either **INSERT** or **REPLACE**, a menu of templates is available for selection; the user can select the 'source' of the operation to be a language template from the corresponding menu, type in text at that point, or select an existing program structure.

Since the **REPLACE** command does not require an option specification, the menu table for it can be separated from that for the **INSERT** command. For both the commands the number of menu tables are roughly equal to the number of primary domains in the cursor classification. The tables used have a maximum of three dimensions. As before, the small size of the tables allow for modifications to be localized to small data structures.

If the user instead selects an existing program structure, the necessary test before calling the editing action routine is to check for compatibility. This test could be

performed in two ways. In one method, the tables that were constructed to supply the choices from the available language structure templates could be used. This is apparently possible because these structures are of the forms that are applicable for the operation of interest at that point. For example, for an **INSERT-Around** operation on a statement, the menu items would include the templates for all statements of the *container* type. Thus if instead a language structure is chosen to be inserted around, it may be checked for compatibility by testing whether the form of the structure is included in the set of the available templates. In a different approach, compatibility can be checked by examining the structured cursor values of the source and destination structures at varying levels of precision, as determined by the situation. For example, with an **INSERT-After** operation, if the domain of the destination is **Statements,** then compatibility test requires testing only that the domain of the source is also **Statements.** However, if the operation were to be **INSERT-Around,** then the test also requires checking that the further specification of the source operand shows it to be of **Container-Statement** type. In the implementation for the present editor the second method was chosen even though the first method may have meant savings in space and time by possibly using a simple set membership test. The reason for this is the existence of *grouped* structures. It is not possible to represent such structures as templates because they could be of a variable number of language structures. However their behavior is generally similar to those templates that act as containers. Thus to avoid incorporating 'invisible' templates the compatibility checking scheme using cursor specification of both the operands has been adopted. The performance is not necessarily compromised because structures with widely differing cursor values can be found to be non-compatible with a superficial, or low-precision check.

# Chapter 5: Contextual Issues in a Fragment-Based Program Editor

Programming language definition includes many aspects that are not described fully by a context-free grammar. Among these are the contextual constraints of a programming language's syntax. These are also termed as the semantics, or the static semantic rules of the language. These terms will be used interchangeably in this chapter. Example contextual constraints of a language are the scope rules and the type rules. In a regular compiler, these aspects of a programming language are enforced by a mechanism that is different from the context-free parser that is used in the syntax analysis phase. In a similar manner, in a language based program editor, the contextual constraints in the language rules will be enforced by mechanisms apart from those used for enforcing the context-free grammar. This chapter provides a description of the tasks involved in the implementation of such a mechanism, as well as providing a survey of some of the recent research contributions in this area.

## 5.1. The Incremental Nature of the Problem

In editing via a structure editor like the one for MUPE-2, the editor operates on an intermediate representation between the text of a program and the code that executes on a computer system. The intermediate representation used in this editor is the Abstract Syntax Tree. The user operates on this representation via some *view*. This view is representative of the internal structures and could be a 'prettyprinted' unparsed display or the static scoping display of the structures. The internal structures accessed by the Editor provide complete information on the adherence or otherwise to the language rules. The user is notified of any conflicts of the structures with the language rules, in the view that is extracted from the internal representation.

During the course of editing the user operates on the internal structures that, as a result, continually undergo changes. Thus, the mechanisms for enforcing the contextual constraints have to deal with dynamic structures. In a conventional editor-compiler system the editing and compiling are strictly disjoint activities in the whole. This implies that the activities deal with the structures as a whole and do not take advantage of the fact that the modifications are mainly small changes and hence may not require redoing most of the parsing. However this is not possible in the text editors because the products of parsing, namely the derived structures, are not handled by such editors. To illustrate the nature of the problem that arises in such language editors, presented below is an example that best represents the problem.

For programming languages from the Algol family, definitions and usages of identifiers are located in productions that are remote to each other in the derivation tree. The conventional multi-pass compiler would use an earlier pass to set up the contexts that are needed for performing the contextual analysis in the usages of the identifiers Since in a language based editor it is possible to modify the definitions, and hence the context, at any instance of time, the identifier usages should also be correspondingly rechecked for consistency with the modified context. The simplistic approach would be to *reparse* the program fragment to obtain the new context, as is done in disjoint editor-compiler activities. However, in an interactive editor such reparsing would have degrading effects on the response of the editor. Here, prior to the editing operation the semantic consistency or otherwise can be readily identified Thus after a structural modification due to editing it is only required to identify where and if any of the program fragment's existing structures are affected. This amounts to major savings in time by avoiding a complete reanalysis for semantic consistency.

Thus the need for an incremental semantic checking mechanism is of utmost importance in language based editors. The problem can be better stated as the following. The internal representation is a *semantic* structure that is obtained by 'decorating' the derivation tree with semantic (contextual) information. (The resulting structure may not strictly be a tree any longer.) At any time, the semantic structure should either satisfy all the semantic constraints, or if it does not, it should notify the user of such a condition, if

a view of the structure is available. After an editing operation, the contextual analyzer attempts to reestablish the consistency of the semantic information in an incremental manner. Those values of semantic information that indicate violation of the contextual constraints are used for indicating semantic errors or *cautions* (refer section 5.4) in the program display In some situations the contextual analyzer could also be used to perform structural modifications to revert the structures back to a consistent earlier state.

The addition of contextual constraints in language based editors has been a popular topic of research in recent years. This popularity has been mainly due to research in editor generators like the Synthesizer Generator[Reps84], Editor Allen Poe[Johns84], PECAN[Reiss84b] and ALOE[AmbKE84]. These systems allow the user to specify the syntactic and semantic constraints of a language in order to **generate** an editor based on the language. While MUPE-2 does not support an editor generator, it is instructive to note the specification schemes as well as the implementation techniques used in editor generator systems.

## 5.2. The Attribute Grammar Approach.

The attribute grammar (AG) approach for specifying language rules has been used in a number of compiler-writing systems like MUG2[GanRW77], and GAG[KasHZ82]. AGs have the power as well as the simplicity in its ability to assign the context-sensitiveness, or static semantics to context-free descriptions of languages. The use of AGs in specifying the static semantics of a language in the generation of language-based editors was first made in the Synthesizer Generator.

An AG is an extension of a context-free grammar $G = (V_N, V_T, P, S)$ consisting of nonterminals, terminals, productions, and initial nonterminal, respectively. To each symbol $F$ of $V_N$ there is associated a finite set *IN(F)* of *inherited attributes* and a finite set *SY(F)* of *synthesized attributes*. The sets *IN(F)* and *SY(F)* are usually assumed to be disjoint, and that S has no inherited attributes. The inherited attributes of $F$ are those attributes that are defined when $F$ appears in the right-hand side of a production. Similarly, the synthesized attributes are defined with $F$ on the left-hand side of a production.

For each production $p \in P$,

$$p\colon F_0 \rightarrow v_0 F_1 v_1 \ \ldots\ v_{k-1} F_k v_k, \quad F_j \in V_N,\ v_j \in V_T^* \text{ for } 0 \leq j \leq k,$$

there is associated a set of *semantic functions*. For each $\alpha$ in $SY(F_0)$ *there is a semantic function* $f_{0\alpha}$ of functionality $D_{\alpha 1}X\ldots XD_{\alpha m} \rightarrow D_\alpha$. Similarly for each $\alpha$ in $IN(F_j)$ $0 \leq j \leq k$, there is a semantic function $f_{j\alpha}$ of the same functionality. $m$ and $\alpha_i$ depend on $\alpha$ and j. Each $\alpha_i$ is an attribute of either $IN(F_0)$ or $SY(F_l)$ for some $l$, $1 \leq l \leq k$. The semantic functions are used to assign meanings to derivation trees. Consider a derivation tree $t$ and a node $n$ in $t$. Let

$$p\colon F_0 \rightarrow v_0 F_1 v_1 \ \ldots\ v_{k-1} F_k v_k,$$

be the production applied at $n$. For each $\alpha$ in $SY(F_0)$ the function $f_{0\alpha} \cdot D_{\alpha 1}X\ldots XD_{\alpha m} \rightarrow D_\alpha$ associated with $p$ can be used to determine the value of $\alpha$ at $n$ when the values of all the attributes $\alpha_1, \ldots, \alpha_m$ have been determined. Similarly, for $\alpha$ in $IN(F_j)(1 \leq j \leq k)$ the function $f_{j\alpha}$ associated with $p$ is used to determine the value of $\alpha$ at the jth child of $n$. If it is possible to determine the values of all attributes of any node in $t$ then the *meaning* of $t$ will be $t$ decorated with these values.

An AG is now presented as an example. The underlying grammar has five nonterminals with attributes

| | | | | |
|---|---|---|---|---|
| IN(S)=∅ | IN(defs)=∅ | IN(typedecln)=∅ | IN(ident)=∅ | IN(uses)={symtab} |
| SY(S)=∅ | SY(defs)={symtab} | SY(typedecln)={typeval} | SY(ident)={stringrep} | SY(uses)={typeclash} |

The grammar with the semantic functions is as below. The notation sym.attr represents attribute attr of symbol sym, and Func(.) denotes a function over attribute values.

| | |
|---|---|
| S→defs';'uses | uses.symtab=defs.symtab |
| defs→typedecln ident';'defs | $defs_1$.symtab=Decln($defs_2$.symtab,typedecln.typeval,ident.stringrep) |
| defs→$\epsilon$ | defs.symtab=empty |
| typedecln→CHAR | typedecln.typeval=character |
| typedecln→INT | typedecln.typeval=integer |
| uses→ident':='ident';'uses | $uses_2$.symtab=$uses_1$.symtab |
| | $uses_1$.typeclash=(Type($uses_1$.symtab,$ident_1$.stringrep) ≠Type($uses_1$.symtab,$ident_2$.stringrep)) |
| uses→$\epsilon$ | |

A derivation tree node of a program fragment describes instances of the attributes of the symbol at that node. A derivation tree 'decorated' with attribute values at its nodes is termed a **semantic** tree. The notion of *consistency* of a semantic tree is primary to the operation of the incremental semantic checking procedure as used in the Synthesizer Generator. A semantic tree is consistent if for all attribute instances, its arguments are available and the value of each attribute instance is equal to the semantic function applied to the arguments.

The basic idea of the AG-based incremental semantic checker is as follows. The semantic checking routine is executed every time a tree editing operation takes place. A tree editing operation always takes place at a single node because any insertion, deletion or replacement in a tree is basically an application of a *pruning* or *grafting* operation at a node in the tree. Thus after an editing operation, if any change in contextual constraints took place then the attribute instances at the node of editing would be *inconsistent*. The incremental semantic checker starts at the node where the editing took place and proceeds by examining those nodes whose attribute instances depend on the initial and subsequent inconsistent attribute instances that arose due to the editing. The major part of the work of the semantic checker is spent in ordering the dependencies of the attribute instances for reevaluation of their values.

An incremental semantic checking scheme is optimal in time if it validates the contextual constraints of only those nodes in the tree that are affected by any non-local changes. The scheme formulated by Reps [Reps84] is optimal in this sense. However for ordering the nodes for reevaluation, the semantic checking scheme has to maintain dependency graphs among the attribute instances at all nodes of the tree. The resulting storage requirements, however optimized, are enormous and in most cases exceed those for the attribute instances at all nodes of the tree. Furthermore, the implementation of the dependency graphs and the operations on the graphs require an enormous amount of code. For use in a non-generator environment, we have found that the payoffs to using such a scheme are discouraging.

Among the criticisms of AGs for handling incremental semantics are:

(i) They are limited to **checking.** To handle anything that requires side-effects like using default values, or forcing recompilation, extensions are necessary that go beyond the formal applicative nature of the AGs. They are also limited to handling **internal, static semantics.**

(ii) They are a low-level description of the language's semantics. The description of the semantic functions involves writing a substantial amount of code The AG method is dependent on the derivation tree produced by the context-free grammar. As a result the symbol tables 'percolates' up from the definitions and then 'trickles' down to the usages. Hence for a single change in a declaration, the entire symbol table may have to be recomputed.

Thus while the AG approach is attractive in a limited sense, and given its popularity in compiler-writing systems, it did not appear as a viable alternative for implementation in our Editor The next section presents the scheme that was adopted for MUPE-2.

## 5.3. The Identifier Map Approach

The above has been so termed after the use of *Identifier Maps* in the incremental compilation model of Magpie[SchDB84] While the basics of the method developed for MUPE-2 are inspired by the scheme used in Magpie, there are significant differences in the two incremental semantic checking schemes. The Magpie scheme of semantic checking is closely related to the incremental parsing that is used for the purely textual mode of editing that Magpie supports. The incremental parsing is achieved by LL(1) techniques and it makes use of *fragmenting* the program into major syntactic units This has been done because with the LL(1) parsing technique static semantic checking cannot be performed beyond the first syntax error. Thus by analyzing each fragment separately, that is the program in smaller parts, the above disadvantage has reduced effects

However the scheme employed to deal with the program fragments during editing was inspiring to our implementation because MUPE-2 deals with program fragments. But while in Magpie the fragmenting was done by the editor/compiler without, perhaps, the

user being aware it was done, the fragments for MUPE-2 are present because of the user's ability to develop program fragments. To avoid any possible confusion between the term fragment as used in Magpie, with that being used in MUPE-2, the former type will be termed node.

The nodes in a program fragment are basically the procedure elements that make up the static scoping hierarchy in a program fragment. Every node has associated with it an Identifier Map that holds entries for all identifiers referenced within the node. It does not matter whether or not the identifier is declared within the node. In the case that the program fragment is of granularity equal to or smaller than a node, e.g. a sequence of statements, the fragment has a single Identifier Map. For fragments of larger granularity, there would be a collection of Identifier Maps, one for each node.

The Identifier Map at a node represents the binding of symbols to objects and it contains the following information about all identifiers appearing within that node:

(i) Identifier Name.

(ii) Where the identifier is defined. If the declaration occurs within the node, it referred to as a local declaration. If not, it is a non-local declaration and it contains a pointer to the defining node, if any.

(iii) The category of the identifier - whether it is a procedure, variable, type etc.

(iv) The representation and value attributes of the identifier as would be found in a normal symbol table.

(v) The references of the identifier. It contains pointers to the places within the node where the identifier has been referenced. If the identifier is declared locally, it also contains pointers to those nodes that reference the identifier within its body.

There is a significant difference in MUPE-2's Identifier Maps from those used in Magpie, because of the presence of import and export facilities in Modula-2. Each module name imported into a module, say A, has the effect of making the imported symbols locally defined in A. Thus there would be entries in some other modules that treat some symbols to be defined in A, even though they might have been imported into it. Symbols that are not declared locally either point to the node where it is defined, or to an ancestor node

where it is imported.

```
MODULE ModDemo;
    IMPORT InOut;
    VAR value, count: INTEGER;

    MODULE NumberGenerator;
        FROM InOut IMPORT
            WriteString, WriteInt, WriteLn;
        EXPORT WriteVal, NextVal;
        VAR CurVal: INTEGER;

        PROCEDURE WriteVal(val: INTEGER);
        BEGIN
            WriteString("Value is:");
            WriteInt(val, 3);
            WriteLn
        END WriteVal;

        PROCEDURE NextVal(): INTEGER;
        BEGIN
            INC(CurVal);
            RETURN (CurVal)        (* A *)
        END NextVal;

    BEGIN
        CurVal := 0
    END NumberGenerator;

BEGIN
    FOR count := 1 TO 10 DO
        value := NextVal();
        WriteVal(value);
    END
END ModDemo.
```

Figure 5.1- Program Module ModDemo.

| Identifier | How defined | Category | Where referenced |
|---|---|---|---|

ROOT environment

| | | | |
|---|---|---|---|
| Modula-2 Ids | Local | | |
| InOut | Local | def module | Heading of Inout |
| | | | Body of ModDemo |
| ModDemo | Local | prog module | module heading |

ModDemo environment

| | | | |
|---|---|---|---|
| NumberGenerator | Local | Module | Heading of NumberGenerator |
| value | Local | Variable | Body of ModDemo |
| count | Local | Variable | Body of ModDemo |
| NextVal | Nonlocal@NumberGenerator | Procedure | Body of ModDemo |
| WriteVal | Nonlocal@NumberGenerator | Procedure | Body of ModDemo |
| InOut | Nonlocal@ROOT | Def module | Import list of ModDemo |
| WriteString | Nonlocal@ROOT | Procedure | Body of NumberGenerator |
| WriteInt | Nonlocal@ROOT | Procedure | Body of NumberGenerator |
| WriteLn | Nonlocal@ROOT | Procedure | Body of NumberGenerator |
| INTEGER | Nonlocal@ROOT | Type | Decln of ModDemo |

NumberGenerator environment

| | | | |
|---|---|---|---|
| InOut | Nonlocal@ModDemo | Def module | Import list of NumberGenerator |
| WriteString | Nonlocal@ModDemo | Procedure | Import list of NumberGenerator |
| WriteInt | Nonlocal@ModDemo | Procedure | Import list of NumberGenerator |
| WriteLn | Nonlocal@ModDemo | Procedure | Import list of NumberGenerator |
| WriteVal | Local | Procedure | Body of ModDemo |
| | | | Export list of NumberGenerator |
| NextVal | Local | Procedure | Body of ModDemo |
| | | | Export list of NumberGenerator |
| CurVal | Local | Variable | Decln of NumberGenerator |
| | | | Body of NumberGenerator |
| | | | Body of NextVal |
| INTEGER | Nonlocal@ROOT | Type | Decln of NumberGenerator |

WriteVal environment

| | | | |
|---|---|---|---|
| val | Local | Variable | Param. list of WriteVal |
| | | | Body of WriteVal |
| WriteString | Nonlocal@NumberGenerator | Procedure | Body of WriteVal |
| WriteInt | Nonlocal@NumberGenerator | Procedure | Body of WriteVal |
| WriteLn | Nonlocal@NumberGenerator | Procedure | Body of WriteVal |
| INTEGER | Nonlocal@ROOT | Type | Param list of WriteVal |

NextVal environment

| | | | |
|---|---|---|---|
| INC | Nonlocal@ROOT | Procedure | Body of NextVal |
| CurVal | Nonlocal@NumberGenerator | Variable | Body of NextVal |
| RETURN | Nonlocal@ROOT | Procedure | Body of NextVal |
| INTEGER | Nonlocal@ROOT | Type | Proc heading of NextVal |

Figure 5.2- The Identifier Maps of the Program Module ModDemo

An example of a program fragment's associated Identifier Maps is shown in Figure 5.2. The figure shows the Identifier Maps at each of the nodes of the program fragment shown in Figure 5.1. The columns in the maps show the values of the information items i, ii, iii and v, as enumerated above. The ROOT environment for this fragment hold the declarations of the standard identifiers of the language, the modules that are imported by the program fragment, and the program fragment itself. Within a module, the effect of an import list is to make available the identifiers that are now visible to the module and the nodes that inherit the identifiers either implicitly, or via import lists in them. However the entry for 'where-defined' shows such identifiers to be 'non-local' For nested modules that import identifiers *defined* in the ROOT environment, the 'where-defined' entry points only to its immediate parent module. This is because nested modules do not, strictly, have access to the ROOT environment (except for the standard identifiers, which are unalterable).

The reference entries in the Identifier Maps enable the incremental semantic checking routine to be performed in an efficient manner. Whenever the definition of an identifier changes, the reference list identifies the usages of the identifier that require reevaluation of the contextual constraint consistency. While such checking of changes to definitions appears to have an advantage over the *'transport'* of symbol tables with the pure attribute grammar approach, there is a price to be paid in keeping the reference list consistent.

An example of such a situation is illustrated with reference to the module shown in Figure 5.1. Suppose the statement at (*A*) which uses Curval is deleted (perhaps as a prelude to some modifications to the procedure body of NextVal). This means that the

reference list that is maintained at NumberGenerator has to be updated to reflect the change. Without the use of back pointers that have a premium on space requirements, it is necessary for the semantic checking routine to first locate the nearest definition and traverse the reference list to make the necessary deletions in it. Such a task is not required in the AG approach because usage nodes do not have any semantically dependent nodes that need updating. Another situation where reference list updating is required is when the scope of an identifier is modified by an inner declaration that hides some outer declaration in the inner nodes. However with the use of the local Identifier Maps the situation is not as bad as it might originally seem. Whenever a new declaration of an identifier is made, a search is done for any occurrence of the identifier in the nodes that are within the static scope of the new declaration. For any identifiers found whose definitions occur outside this scope, their references at those nodes need to be transferred to the new declaration's reference list. The presence of Identifier Maps helps in identifying quickly the references in the inner nodes that need contextual constraint reevaluation. Otherwise all the inner nodes would have to be semantically analyzed in their entirety.

Thus the adoption of the Identifier Maps helps in implementing an incremental semantic checker which is very essential for an interactive language based editor such as the one in MUPE-2. The next section looks at the problem of incomplete information that arises when the editor has to operate with fragments that are not *compilation* units in the usual sense.

## 5.4. Incomplete Information in Fragments

The fragments that are found in MUPE-2 are arbitrary, but well-formed by environment rules, parts of a program. For example, a fragment could be a sequence of statements, a procedure declaration or a whole program. In general, it is a sentential form of some non-terminal of the language. The fragments could contain incomplete components by the presence of templates that are not fully expanded. The use of fragments in MUPE-2 allows bottom-up system development by combining fragments. The fragments themselves can be constructed top-down by refinement of templates.

The realization of a program specification is achieved by the use of declarations and the main program body. The use of fragments allows the user to build and essentially study individual components of a program in isolation. As a result these components present a situation where the contextual constraint analyzer has to be necessarily 'permissive' in enforcing the constraints. In essence, this means that the prototyping nature of the environment should allow incompletely developed program fragments to be conveniently used in building blocks manner, without swamping the user with error messages during the development.

To our knowledge only one existing system has addressed the problem of allowing the use of program fragments as a normal programming activity in language-based editors. It is the Programming System Generator PSG [BahSn85] that generates language-based interactive programming environments from formal language definitions. Presented below are the requirements for the contextual constraint analyzer as formulated by the authors of PSG.

1. The checking algorithm of the context conditions must be applicable to each fragment of the language.
2. The checking algorithm must detect error situations immediately.
3. The algorithm must compute all the type information, which is valid for all extensions to correct programs.
4. The type information of a composed fragment can be evaluated from the type information of the sub-fragments.

Requirement 1 essentially deals with the new situation that arises when fragments are used as programming entities. The checking algorithm should now be capable of handling incomplete information in fragments that cover the wide spectrum of non-terminals of the language. Error situations arise in a program fragment if, firstly, the fragment is a complete program with semantic errors in the usual sense; or, if it is a fragment of smaller granularity that shows inherent contextual conflicts that would prevent embedding the fragment in a correct program. As an example of the latter case, consider a fragment that contains statements without any accompanying declarations. In such a

fragment, it should not be possible to use an identifier both as a procedure call and as a variable that is assigned a value. Thus requirement 2 addresses the need for such a checker that provides error notifications even with incomplete information. Requirement 3 is in the domain of type inferencing. While it is useful for composing fragments from existing ones, it could also be used to aid the user in selecting declarations from the possible set based on the usages of the identifiers. Type inferencing mechanisms have been used in programming environments like PSG and Ape [Levy84]. Requirement 4 guarantees syntax oriented efficient evaluation in the contextual constraints. The subfragments in the composition of a larger fragment contain all the relevant information in their individual *property lists* (or Identifier Maps). It should not be necessary to parse the resulting fragment to derive its property list. Hence the information extraction should be done in an incremental manner.

The use of Identifier Maps has enabled the design of the Editor's contextual constraint analyzer, which meets the above requirements except for third one. We do not yet use any type inferencing because of the complexities involved in dealing with such a wide spectrum of fragment types. Incomplete information for contextual analysis in any fragment has to be handled by the Editor in a way which makes a distinction between an incomplete program fragment and a complete program fragment. This is necessary because the notification of an error in one case may not imply the same in the other. This situation arises in checking whether a symbol is declared or not. In a complete program fragment, any usage of symbols without their declarations is an error and the user is notified as such. However when dealing with incomplete program fragments, symbols should be allowed for use without declaring since the complete context is not necessarily available. The way this is being handled in our Editor is in distinguishing the Identifier Map at the outermost scope.

In the contextual analysis when a symbol usage is encountered, an entry is made in the local *scope* (Identifier Map) for the symbol's declaration. If it was the first usage of the symbol in that scope, there would be no reference to that symbol's definition. This initiates a search, up the environment tree, to locate the definition. If the search is successful, the subsequent action is dependent on the type of the program fragment.

If the fragment is complete, i.e. a regular compilation unit, then the earlier usage of the symbol was an **error** and the user is notified as such. However if the fragment is not complete, then the current context is not the complete context. Hence it is inappropriate to indicate a semantic error for such an occurrence. Instead, it is better to indicate a semantic **caution.** Internally, this is handled by making a local definition of the symbol in the scope it was used. It is assigned an **open** type. Such a type always passes the type compatibility test. If the symbol is any form of procedure then the formal parameter list is another 'open' interpretation. Note that such assignments are not similar to the default types like **integer** that are assigned to undeclared symbols in conventional compilers. If we were to use such a scheme, then the bottom-up fragment construction method would not hold because of possible conflicts with the 'default' assignments. The **open** type assignments are also used in instances of declarations that are not fully expanded, as in

VAR foo · <type>

The use of **open** is perhaps less restrictive and, as a result, less safe than in using type schemas. However use of the category attribute meets requirement 3 'partly'. There is an investigation in progress for using type inferencing in MUPE-2.

## 5.5. Interfragment Operations and Dependencies

This section examines some of the issues that arise with operations that involve more than one fragment. In the discussions so far, it had been tacitly assumed that the Editor operated on only one fragment. However in MUPE-2 the more general situation is with more than a single fragment for multi-operand operations. The ability to handle multiple fragments by the Editor also arises due to the ability to make use of *separate compilation* facility with strong type checking across modules

The requirement for the Editor to be able to operate with multiple fragments makes it necessary to transfer structures between fragments. Such a transfer could occur as either a *copy,* where there is a no deletion of structures, or it could be as a *move,* where there is a deletion of a structure in one fragment and its insertion in another fragment There does not seem to be any existing system that provide such features in an editor.

What has normally been available in textual editors was some form of *clipboard* facility, which provides a temporary repository for moving objects. However such transfer mechanisms do not provide the ease of operation or the protection that the MUPE-2 Editor seeks to provide. The main reason why such features are attractive for the MUPE-2 context is because of the possibility of having displays of many fragments at a time. This has mainly arisen due to the emergence of *windows* in display technology [Hopgo86]. Such a user interface makes it possible to the user to operate on multiple fragments that are open on the screen. However to accommodate such features requires the Editor to be designed to keep track of the 'last cursor' and 'last operation' with each fragment. This is an additional requirement to what are normally supported in present window managers and it entails providing the editor with window management features.

To explain the working of the contextual analyzer in such a situation, we first show the user actions to achieve such a transfer. A scenario for such an operation is sketched in Figure 5.3. Initially, the active fragment is Fragment #68. Its cursor is the **while** structure denoted A. The user selects the **MOVE** command, which is qualified further by **Before**. The menu of templates for this operation is presented for selection, but the user instead decides to move a structure from a different fragment, #102. To achieve this the user selects, via the pointing device, Fragment #102 to be the active fragment. Fragment #68 loses its active fragment status and the incomplete operation initiated at structure A is *frozen*. The user makes the cursor in the current active fragment (#102) to be the desired structure B by grouping the two statements as such. Having done this, the user selects the frozen operation in Fragment #68. This makes the last structure in Fragment #102 to be a candidate for the operation resumed in Fragment #68. The candidacy validation is carried out as explained in Chapter 4. After the validation, the designated structure at Fragment #102 has to be extracted, its Identifier Map information constructed and the modified Fragment #102 should have its Identifier Maps updated. With the temporary fragment thus formed, it is inserted in Fragment #68. The Identifier Maps in this fragment are now updated. Thus the contextual analyzer has to work at two fragments to affect the single operation initiated by the user.

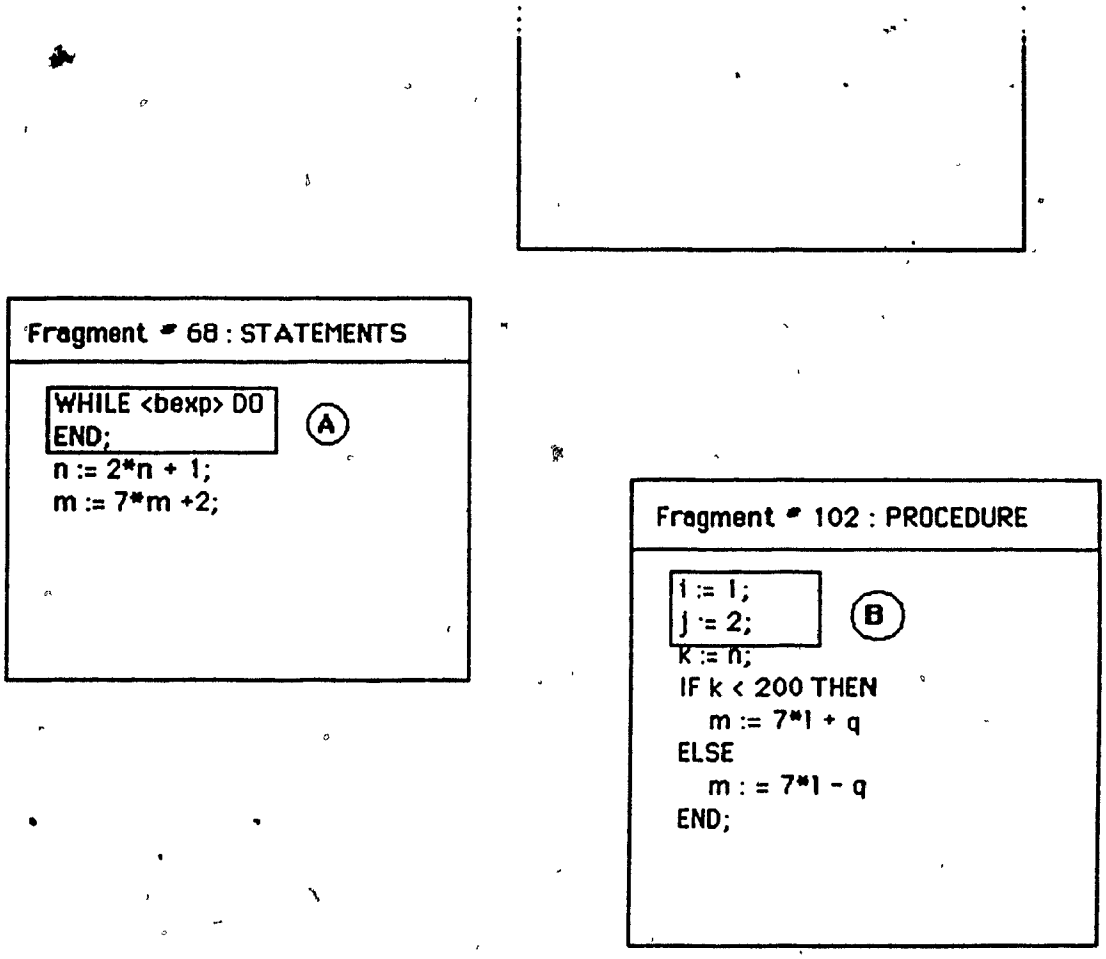


Figure 5.3- Scenario of Interfragment Operations.

What is of prime consideration in the implementation of the above operation is the process of deriving the Identifier Map values of a chosen structure from its parent structure and its Identifier Map. It is however advantageous to extract the required information from the existing Identifier Maps and hence avoid a contextual analysis for the structure extracted. Incremental operations are always better for response time considerations in interactive systems. With the usage of Identifier Maps it is necessary to identify the symbols that occur in the subtree that is the object of transfer. These can be identified by traversing the leaves of the concerned subtree. The corresponding entries in the Identifier Map will provide information about the symbol's category, its type and others. But extracting them in their entirety may not reflect the true status of the derived

fragment from the subtree. This is mainly for definitions that are not present in the extracted subtree. Such a case is especially true when considering types that are subtypes (subranges) of a larger type. In Figure 5.3, consider the variables i and j to have the CARDINAL data type in fragment #102. However, it is probable that fragment #68 already has these identifiers declared as variables of INTEGER data type. Retaining the more restrictive CARDINAL type would create a conflict while inserting that structure in fragment #68. In such a situation the symbol's type loses its resolution and takes on the **open** type as was explained earlier. Thus when the extracted structure is inserted in the destination position, it is not necessarily tied to the earlier types of the symbols, but it utilizes such invariant information like the category, since it is based on the syntactic usage.

## 5.6. Issues in Interface Control

Another point that arises in such an editor is in the support of modularity and data abstraction. Languages like Modula-2 and Ada offer the concept of a **module** or **package** to encourage modularity and data abstraction. However these languages have been designed for textual processing, and the normal declaratory syntax has been extended to support visibility control for the above features. As a result the emphasis on the complexities introduced in the syntax distracts the programmer from the primary issue of visibility control. The structured editor can provide substantial help to the user by providing editing operations that are meant specifically for such aspects. The need for such facilities is particularly felt for programming teams where the user and the definer of a module are not the same. The responsibilities for syntactic correctness is now distributed. In the following are descriptions of the MUPE-2 Editor with regard to extending the scope of a structured editor for such purposes It should be noted that the editor Yggdrasil [Capli85] provides similar extensions to structured editors. However, the editing operations in that editor are basically carried out in 'name definition' windows, without the textual context of the program body. This isolation could prove to be too powerful to control. Furthermore the Yggdrasil editor has been designed for extending existing conventional languages with data abstraction and modularity without changing

their syntax. The MUPE-2 Editor is designed for Modula-2 where modularity and data abstraction are part of the language and the Editor operations introduced here help using these features.

Modula-2 motivates the partitioning of a program into modules. Each module can contain constants, variables, procedures and perhaps types. Objects such as these that are declared in Module A can be referred to by another Module B if there is a **provision** (export) in, say, Module A, and a **requisition** (import) in Module B. Interface control constitutes this specification and control of the interactions among entities in different modules. Modula-2 supports separate compilation. This means that a program can import objects from modules that do not need to be compiled together with the main program. At the compilation time of this importing program the description of the imported objects should be available. However the details of such objects are not essential and each module provides an abstraction. It is also a means of **protection** by the ability to *hide,* via the abstractions. Thus Modula-2 provides a textual separation of the essentials from the details. The essentials provide the descriptions of the objects that are used by other modules; the details constitute the parts that are hidden and hence **private.** This divides modules into two parts: a **definition module,** which describes the objects that can be used by other modules, and an **implementation module,** which contains the bodies of the objects, as well as other objects that can be used only by the implementation modules. The objects visible outside the module are enumerated explicitly by an **export list.** The two parts are compiled separately and are called *compilation units.*

The above concepts having been developed for the traditional batch compilers do not provide the same freedom and advantages when used with structured editors. These editors operate on an intermediate representation of a program that is usually some form of a program tree. However the definition modules do not reflect any such tree structure. In a batch compiler the result of compiling a definition module is a **symbol file,** which is used for cross-module type-checking, etc. So to use a common structure editor for all program fragments means having to implant some form of tree representation on the definition modules. Since it distorts the 'natural' view of the definition modules, the

MUPE-2 design breaks away from the dichotomy between definition and implementation modules present in the textual models.

The MUPE-2 representation of a definition and implementation module pair is as a single structure, a *capsule*. This structure is representative of any program module. It follows the structural aspects of any single module. However since the purpose of the textual separation was to provide a separation between the external description of a module (the abstraction) from the inner details (the implementation), the user interface for the Editor maintains it. Figure 5.4 is an illustration of such a view.

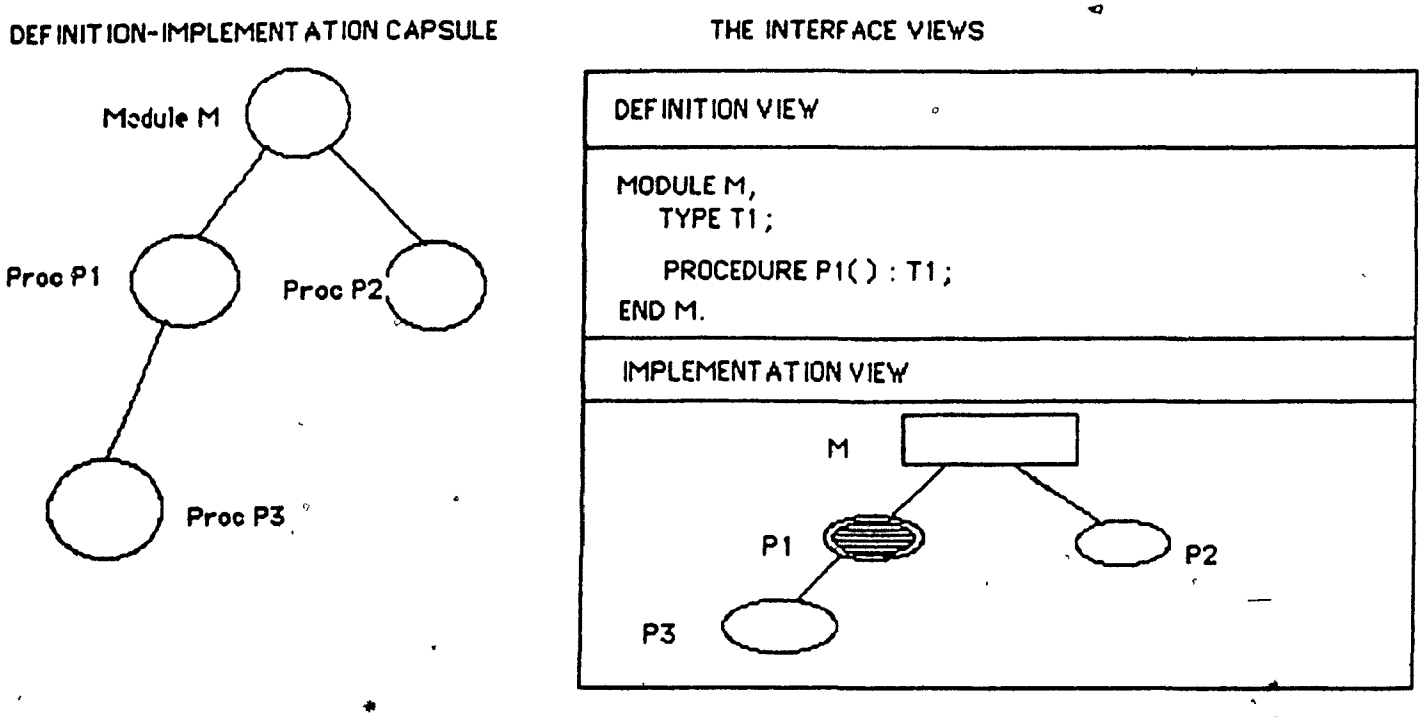


**Figure 5.4- Interface Views of Definition and Implementation Module Pair.**

The view that is presented to the user depends on the user's access privileges to the views of the module. For users with access to only the definition, they are provided with a window of the descriptions of the exported objects. There is no corresponding view of the implementation of the module. There is no editing cursor in the definition view. For the more general situation where the user possesses implementation access, the Editor provides two views of the module. One is the definition view as described above, and the other view is of the implementation. The editing operations are available in the capsule as a whole.

The present implementation of visibility control reflects the updated Modula-2 language report where all objects defined in the definition module are exported; in the earlier language report, strictly only those that appear in the export list were exported. Each object declaration in the main body of the module has an associated attribute that indicates the visibility status. The status can be one of the following: hidden, opaque export or full export. The visibility attribute is recorded in the module's outermost Identifier Map. This map is accessible from other modules for purposes of cross-module checking. At the user interface, exported objects are displayed in the definition view Since the editing operations are present only in the implementation view, the exported objects are shown in there too, but distinguished from the non-exported objects by some means of *high-lighting*. All the objects in the level from where objects can be exported, allow editing operations on the objects that can change their visibility status values among the three permissible values.

Thus designing an editor knowledgeable in the interface control properties provides freedom to the user from the restrictive nature of language extensions in declaratory syntax. The visibility or interface control is normally a distinct part from the *computation* that the rest of the program is concerned with. Hence the availability of visibility control features in the editor operations are necessary to improve the programming process. The decision to maintain a single structure for the definition and implementation parts of a module was influenced by the editing operations that are possible with structures of tree types. As well, the requirements for cross-module type checking for languages like Modula-2, decided that the intermediate representation for the definition-part of a module should be nothing more than a symbol table.

# Chapter 6: Conclusions

This thesis has examined the issues involved in designing a Fragment-based Program Editor. Such an editor is the central part of a programming environment that supports editing, compiling, executing and debugging of program fragments. This thesis has concentrated on the editing aspects of such an environment that is under development, the McGill University Programming Environment (MUPE-2). Within a highly integrated environment the editor provides the user interface and consequently decides many of the system's architectural issues.

The MUPE-2 environment has been designed to program with fragments of the programming language Modula-2. The concept of Fragtypes that is central to this environment, and the relevant fragtypes for Modula-2, were presented in this thesis. While Modula-2 provides for the development of software through modules, it has been found inadequate in the degree of reusability that an individual programmer would be interested in. MUPE-2's typed fragments addresses this shortcoming and proposes that an integrated environment that is built on the concept of fragments would be an answer in increasing programmer productivity.

MUPE-2 has been aimed to be operational with graphics and a pointing device at the interface, in addition to the normal keyboard. Consequently, the command and response language offers significant differences from those found in systems based on text and lines. The model that is used in the specification of operations with such an interface has been presented and it plays a significant role in the implementation of the editing rules.

Portability and adaptability were set as important design goals in MUPE-2. In the editor design that has been presented in this thesis, these goals influenced many of the design decisions. Powerful present-day engineering workstations are the target systems

for MUPE-2. However these workstations feature many non-standardized utilities that are essential for implementation. As a result, the architecture of this environment includes a layer, called the Screen Manager, that encapsulates the terminal and other workstation dependencies. The present design of the Screen Manager also makes it possible to achieve a distributed implementation of the environment. In such a scheme, the Screen Manager which is responsible for the display and user input, could be physically separated from the rest of the system that performs the tasks of menu generation, compatibility checking, incremental compilation and others.

User input in language-based editors can be in two basic models. One is the template model where the input is via templates of the language. In the other model, the user inputs text that has to be subsequently parsed. While the MUPE-2 editor's goal is to allow both forms of input, this thesis has concentrated only on the template form of user input. In such a case, the role of the parser is played in determining what are the templates that are allowable to be input at an arbitrary point in the program. The scheme that has been presented in this thesis shows how the language's syntax rules can be used to allow a table-driven implementation that derives the set of legal templates at arbitrary points of the program. A classification model of language structures is shown that permits an efficient organization of tables. Table driven implementations have been favored at all stages of MUPE-2's implementation because of the degree of adaptability possible from them.

Apart from maintaining the developed software in accordance to the language's syntax rules, a language based editor also has to enforce the contextual constraints in the language rules. Such constraints are typically the scope and type rules of the language. The distinctive features of these rules are that they allow editing operations to affect the validity of structures that may not be in the immediate neighborhood of the location of the editing operation. Attribute grammars has been applied to solve this 'inheritance' problem in an incremental manner within language based editors. However the solutions, while optimal in time, have enormous space requirements. The scheme presented in this thesis for use in the MUPE-2 editor, employs a model that also promises to be useful in considering the visibility control features of Modula-2, and the multi-fragment operations

permitted in MUPE-2.

Modularity and data abstraction are perhaps the most important features of Modula-2. However in a structure based editing environment, the specified compilation model of the language for textual environment does not provide a desirable user interface. Visibility control within a structure based environment is better achieved by application of visibility control operations on objects rather than being affected by declaratory program elements. The usage of such operations, and the user interface adopted for this purpose, has been presented in this thesis.

The goal of MUPE-2 is to provide a well integrated environment for the development of software. Within this environment it would be possible to specify, edit, compile, execute, debug and document software. This thesis has presented the implementation scheme for the language oriented language editor that is at the heart of the system. The implementation is not yet complete and consequently, it is difficult to evaluate the design so presented.

*"Interactive systems are meant to be experienced, not talked about." - Anon.*

## References

[AmbKE84] V. Ambiola, G. E. Kaiser, and R. J. Ellison, An Action Routine Model for ALOE, Tech. Report CMU-CS-84-156, Carnegie-Mellon University, August 1984.

[ArchC81] J. Archer, and R. Conway, COPE: A Cooperative Programming Environment, Tech. Report 81-459, Cornell University, 1981.

[BahSN85] R. Bahlke, and G. Snelting, The PSG - Programming System Generator, Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments, *ACM SIGPLAN Notices*, Vol. 20, No. 7, July 1985, pp.28-33.

[BecBP82] A. Bechtolsheim, F. Baskett, and V. Pratt, The SUN Workstation Architecture, Tech. Report 229, CSL, EECS, Stanford University, March 1982.

[BuxtD81] J. N. Buxton, and L. E. Druffel, Requirements for an Ada Programming Support Environment: Rationale for STONEMAN, *Software Engineering Environments*, ed. H. Hunke, North-Holland, 1981, pp.319-330.

[CapHo86] M. Caplinger, and R. Hood, An Incremental Unparser for Structured Editors, *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, 1986, pp.65-74.

[Capli85] M. Caplinger, Structured Editor Support for Modularity and Data Abstraction, *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments,* ACM SIGPLAN Notices, Vol. 20, No. 7, July 1985, pp.140-147.

[DelMS84] N. M. Delisle, D .E. Menicosy, and M. D. Schwartz, Viewing a Programming Environment as a Single Tool, *Proceedings of the ACM SIGPLAN/SIGSOFT Software Engineering Symposium on Practical Environments,* ACM SIGPLAN Notices, Vol. 19, No. 5, 1984, pp.49-56.

[DeReK76] F. DeRemer, and H. Kron, Programming-in-the-Large Versus Programming-in-the-Small, *IEEE Transactions on Software Engineering,* Vol. SE-2, No. 2, June 1976, pp. 80-86.

[Donze80] V. Donzeau-Gouge, et al., Programming Environments Based on Structure Editors: The Mentor Experience, Tech. Report 26, INRIA, May 1980.

[Engin85] Engineering Computer Center, Arizona State University, Window Management System, September 1985.

[FisPS84] C. N. Fisher, G. F. Johnson, J. Mauney, A. Pal, and D .L. Stock, The POE Language-Based Editor Project, *Proceedings of the ACM SIGPLAN/SIGSOFT Software Engineering Symposium on Practical Environments,* ACM SIGPLAN Notices, Vol. 19, No. 5, 1984, pp.21-29.

[Fritz84] P. Fritzson, Towards a Distributed Programming Environment Based on Incremental Compilation, Research Report, Department of Computer and Information Sciences, Linköping Univ., Sweden, 1984.

[Gasne83] E. R. Gasner, et al., SYNED - A Language-Based Editor for an Interactive Programming Environment, *Digest of Papers Spring Compcon 83*, 1983, pp.406-410.

[GanRW77] H. Ganzinger, K. Ropken, and R. Wilhelm, Automatic Generation of Optimizing Multipass Compilers, *Proceedings of the IFIP Congress 77*, Elsiever North-Holland, 1977, pp. 535-540.

[Gentl81] W. M. Gentleman, Message Passing Between Sequential Processes: The Reply Primitive and the Adminstrator Concept, *Software Practice and Experience*, Vol. 11, 1981, pp.435-466.

[Hanse75] P. Brinch Hansen, The Programming Language Concurrent Pascal, *IEEE Transactions on Software Engineering*, Vol 1, No. 2, June 1975, pp.199-207

[HenhS84] W. Henhapl, and G. Snelting, Context Relations - a Concept for Incremental Context Analysis in Program Fragments, *Proc. 8. GI-Fachtagung Programmiersprachen und Programmentwicklung, Informatik Fachberichte 77*, Springer-Verlag, 1984.

[Herot80] C. F. Herot, et al., A Prototype Spatial Data Management System, *SIGGRAPH 80 Proceedings of the ACM/SIGGRAPH Conference*, 1980, pp.63-70.

[Hoare78] C. A. R. Hoare, Communicating Sequential Processes, *Communications of the ACM*, Vol. 22, No. 6, June 1978, pp.353-368.

[Hopgo86] F. R. A. Hopgood, et al,(eds.), *Methodology of Window Management*, Eurographics Seminars, Springer-Verlag, 1986.

[Johns84] G. F. Johnson, An Approach to Incremental Semantics, Computer Sciences Technical Report No. 547, University of Wisconsin - Madison, July 1984.

[KasHZ82] U. Kastens, B. Hutt, and E. Zimmermann, *GAG, a Practical Compiler Generator*, Lecture Notes in Computer Science, Vol. 141, Springer-Verlag, 1982.

[LantN84] K. A. Lantz, and W. I. Nowicki, Structured Graphics for Distributed Systems, *ACM Transactions on Graphics*, Vol. 3, No. 1, January 1984, pp.23-51.

[LaueN79] H. E. Lauer, and R. M. Needham, On the Duality of Operating System Structures, *Operating System Review*, Vol. 13, No. 2, 1979, pp.3-19.

[Levy84] M. R. Levy, Type Checking, Separate Compilation and Reusability, *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 19, No. 6, June 1984, pp.285-289.

[Madha85] N. H. Madhavji, Operations for Programming in the All, *Proceedings of IEEE 8th International Conference on Software Engineering*, 1985, pp.15-25.

[MadCR85] N. H. Madhavji, S. Choudhury, R. Robson, and N. Friedman, On Commands for an Integrated Programming Environment. *Foundation for Human-Computer Communications*, (eds.) K.Hopper and I.A.Newman, North Holland, 1986, pp. 407-423.

[MadhP85] N. H Madhavji, and L. Pinsonneault, A Colour Coded Scheme for Handling User Interrupts in Non-atomic Programming Operations, *Proceedings, COMPINT 85*, September 1985, pp.111-113.

[MadPT86] N. H. Madhavji, L. Pinsonneault, and K. Toubache, Modula-2/MUPE-2: Language and Environment Interactions. To appear in *IEEE Software Special Issue on Modula-2*, November 1986.

[MetcB76] R. M. Metcalfe, and D. R. Boggs, Ethernet: Distributed Packet Switching for Local Computer Networks, *Communications of the ACM*, Vol. 19, No. 7, July 1976, pp.395-404.

[Nagl83] M. Nagl, An Incremental Programming Support Environment, Tech. Report OSM-I-11, University of Osnabruech, 1983.

[NakYu83] R. Nakajima, and T. Yuasa (eds.), *The IOTA Programming System*, Lecture Notes in Computer Science, No. 160, Springer-Verlag, 1983.

[Nicke84] R. Nickel, The IRIS Workstation, *IEEE Computer Graphics and Applications*, Vol. 4, No. 8, August 1984, pp.30-34.

[Notki85] D. Notkin, et al., Special Issue on the GANDALF Project, *Journal of Systems and Software*, Vol. 5, No. 2, May 1985.

[PfafH85] G. Pfaff, and P. J. W. ten Hagan, *Seeheim Workshop on User Interface Management Systems*, Springer-Verlag, Berlin, 1985.

[Powel83] M. Powell, Modula-2: Good News and Bad News, *Digest of Papers, Spring Compcon 1983*, pp.438-441.

[Reiss84a] S. P. Reiss, Graphical Program Development with PECAN Program Development Systems, *Proceedings of the ACM SIGPLAN/SIGSOFT Software Engineering Symposium on Practical Environments*, ACM

SIGPLAN Notices, Vol. 19, No. 5, 1984, pp.30-41.

[Reiss84b] S. P. Reiss, PECAN: Program Development Systems that Support Multiple Views, *Proceedings of IEEE 7th International Conference on Software Engineering*, 1984, pp.324-333.

[Reps84] T. Reps, *Generating Language-Based Environments*, The MIT Press, 1984.

[SchDB84] M. D. Schwartz, N. D. Delisle, and V. S. Begwani, Incremental Compilation in Magpie, *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 19, No. 6, June 1984, pp.122-131.

[Sewry84] D. A. Sewry, Modula-2 and the Monitor Concept, *SIGPLAN Notices*, Vol. 19, No. 11, November 1984, pp.33-41.

[Smith82] D. C. Smith, et al., Designing the Star User Interface, *Byte*, April 1982.

[TeiMa81] W. Teitelman, and L. Masinter, The Interlisp Programming Environment, *Computer*, Vol. 14, No. 4, April 1981, pp.25-33.

[TeiRe81] T. Teitelbaum, and T. Reps, The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, *Communications of the ACM*, Vol. 24, No. 9, September 1981, pp.563-573

[Unite82] United States Department of Defense, *Reference Manual for the Ada Programming Language*, 1982, AdaTEC Special Publication.

[Wegma80] M. N. Wegman, Parsing for Structured Editors, *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, October 1980, pp. 320-327.

[Wirth85] N. Wirth, *Programming in Modula-2*, Springer-Verlag, 1985.