THE UNIVERSITY OF MANITOBA

# THE DESIGN OF A SOFTWARE ENVIRONMENT FOR THE DEVELOPMENT OF MODULAR PROGRAMS

by:

D. J. F. Hughes

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF MANITOBA

WINNIPEG, MANITOBA

October, 1979

# THE DESIGN OF A SOFTWARE ENVIRONMENT FOR THE DEVELOPMENT OF MODULAR PROGRAMS

BY

DAVID JOHN HUGHES

A dissertation submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

© 1979

## ACKNOWLEDGEMENTS

I would like to acknowledge Dr. C. R. Zarnke, my supervisor, for his advice, suggestions and corrections and Dr. D. D. Cowan for providing me with computing resources at the University of Waterloo.

## DEDICATION

I would like to dedicate this thesis to my wife, Christine, without whose prodding this thesis would never have been completed.

## ABSTRACT

Modular program development is the state-of-the-art methodology for computer project development. As yet, however, there is no programming environment in which this methodology can be used conveniently and safely for the development of computer software projects.

This thesis examines the current typical program development environment from the point of view of the modular program development methodology and points out the weaknesses in the present environment. It then focuses on the interaction of the user with the system and outlines the goals of a software system suited to this methodology.

These goals are then used in the design of such a software system, called DEMOS. DEMOS is an interactive system for the development of programs. It maintains complete control over the inter module interfaces not only at the user level but also at the operating system level. Consequently, it guarantees complete consistency checking during separate compilation of modules and thus encourages modular development. In this environment a module is always developed as a fragment of a larger program. This program, in turn, is con-

sidered as a module in an even larger program which may be a module of the operating system.

It should be noted that this thesis is concerned with the design of the software environment and does not describe a specific implementation of such a system.

## TABLE OF CONTENTS

Table of Contents . . . . . . . . . . . . . . . . . . . . . viii

Table of Figures . . . . . . . . . . . . . . . . . . . . . . xi

1 Introduction . . . . . . . . . . . . . . . . . . . . . . . 1

1.1 Program Development in Present Day Environment . . 3

1.1.1 The Environment . . . . . . . . . . . . . . . . . 3

1.1.2 Program Development Process . . . . . . . . . . 4

1.2 Problems Encountered in Current Environment . . . 7

2 A Modern Program Development Environment . . . . . . 11

2.1 Abstract Types . . . . . . . . . . . . . . . . . . . 11

2.2 Consistency Checking . . . . . . . . . . . . . . . 13

2.3 Separate Compilation . . . . . . . . . . . . . . . 15

2.4 Automatic Consistency Checking . . . . . . . . . . 16

2.5 Design Goals of DEMOS . . . . . . . . . . . . . . 19

2.6 Requirements for the DEMOS Solution . . . . . . . 20

3 Logical Organization of DEMOS . . . . . . . . . . . . 22

3.1 Modules in DEMOS . . . . . . . . . . . . . . . . . 22

3.2 Module Representation . . . . . . . . . . . . . . 24

3.2.1 Components . . . . . . . . . . . . . . . . . . . 26

3.2.1.1 Abstraction Components

3.2.1.2 Realization Components

3.2.1.3 Separate Procedures

3.2.1.4 Module Instances

3.2.2 Component Dependencies . . . . . . . . . . . 47

3.3 Consistency of Module Components . . . . . . . . 50

3.3.1 Consistency and Compilation . . . . . . . . 50

3.3.2 Consistency and Modification . . . . . . . . 54

3.3.3 Compilation for Consistency Checking . . . 58

3.3.3.1 Categories of Dependence

3.3.3.2 Deferred Consistency Checks

3.4 Abstract Machines . . . . . . . . . . . . . . . 67

3.4.1 Machine Representation . . . . . . . . . . . 70

3.4.2 The Development Environment . . . . . . . . 72

3.4.2.1 Machine Development

3.4.2.2 Machine Use

3.4.3 Machines and Address Spaces . . . . . . . . 75

4 Physical Organization of DEMOS . . . . . . . . . . . 79

4.1 Data Management . . . . . . . . . . . . . . . . 79

4.1.1 Data Areas . . . . . . . . . . . . . . . . 79

4.2 Address Management . . . . . . . . . . . . . . . 83

4.2.1 Address Spaces . . . . . . . . . . . . . . 83

4.2.2 Address Allocation . . . . . . . . . . . . 86

4.2.3 Address Resolution . . . . . . . . . . . . 88

4.2.4 Addressing Scheme . . . . . . . . . . . . . 90

4.2.5 User Defined Address Spaces . . . . . . . . . 98
4.3 Memory Management . . . . . . . . . . . . . . . . 104
4.3.1 Memory Spaces . . . . . . . . . . . . . . . 104
4.3.2 Address Decoding . . . . . . . . . . . . . . 107
4.4 Hardware Support . . . . . . . . . . . . . . . . . 114

5 DEMOS in Review . . . . . . . . . . . . . . . . . . . 119
5.1 Satisfaction of the Goals . . . . . . . . . . . 119
5.2 Additional Benefits and Future Development . . . 125

Appendix A Address Space Module Example . . . . . . . . 128
References . . . . . . . . . . . . . . . . . . . . . . . 136

## TABLE OF FIGURES

3.2.a Module Dependence . . . . . . . . . . . . . . . . . . . 25
3.2.1.a Component Groups . . . . . . . . . . . . . . . . . 27
3.2.1.1.a Abstraction Components . . . . . . . . . . . . 30
3.2.1.1.b Abstraction Representation . . . . . . . . . . 34
3.2.1.2.a Realization Components . . . . . . . . . . . . 36
3.2.1.2.b Realization Representation . . . . . . . . . . 41
3.2.1.3.a Separate Procedure Components . . . . . . . . 43
3.2.1.3.b Separate Procedure Representation . . . . . . 45
3.2.1.4.a Data Area Representation . . . . . . . . . . . 47
3.2.2.a Module Component Summary . . . . . . . . . . . . 48
3.2.2.b Component Dependencies Maintained by DEMOS . . 51
3.3.3.2.a Deferred Consistency Checks . . . . . . . . . . 64
3.4.3.a Contained and Separate Instances . . . . . . . . 76
4.1.1.a Data Area Types . . . . . . . . . . . . . . . . . . 82
4.2.4.a Address Resolution . . . . . . . . . . . . . . . . 93
4.3.2.a Address Decoding . . . . . . . . . . . . . . . . . 111

# 1 INTRODUCTION

Modular program development is the state-of-the-art program development methodology. It incorporates the concepts of stepwise refinement [23], structured programming [9], and top down design [17]. The breakdown of a task into subtasks done by chief programmer teams [1] reflects this methodology. Languages have also been developed with this methodology in mind [24]. However, up to now, few attempts have been made to provide a programming environment in which this methodology can be used conveniently and safely for the development of computer software projects [19], [22], [6], [4].

modular programming implies the decomposition of a system into a number of interacting components called modules. The decomposition is done according to a set of guidelines. A number of sets of guidelines have been used including functional decomposition [23] and decomposition by abstract data types [14]. Other decomposition criteria have also been proposed [21]. Throughout this thesis the term modular programming will be used to imply decomposition by abstract data types.

The programming environment includes the hardware (e.g. terminals) and software (e.g. interactive systems) used in program development. The current environment is inadequate mainly in the area of software. The software provided in current program development environments is outdated by the modern program development methodology. It is the interaction between the user and the software system which requires updating to make the program development environment suitable for modular program development.

This thesis, therefore, concentrates on the design of a suitable software system. The designed system, DEMOS (Development Environment for MOdular Systems), integrates the tools of program development into a single system which is oriented towards modular programming. Included in the system are facilities for module abstraction and realization, control of module interaction and resource needs, and testing and modifying of modules. The system is interactive since program development requires constant feedback. The system automates consistency checking and therefore frees the programmer from the error prone task of verifying consistency of interfaces between modules. Module development is done within the environ of an abstract machine which itself contains modules which are in the environ of still other abstract machines which ultimately are machines of the operating system.

The designed system will realize these goals and, at little extra cost, will yield added advantages. Some of these advantages include the easy integration of: a high-level debugging aid which can yield meaningful information, a simulator for modules which have been designed and not yet implemented, and an automated program prover (when this becomes possible).

## 1.1 PROGRAM DEVELOPMENT IN PRESENT DAY ENVIRONMENT

### 1.1.1 THE ENVIRONMENT

Tools in the software system of the typical program development environment include a procedure oriented language and a compiler for it (eg. PL/I, FORTRAN, COBOL), an online editing and job submission (usually to batch) system (eg. TSO), a linkage editor or loader, a job control language processor and system level routines which are used as primitives in program development.

The language and compiler allow compilation of a procedure and perform type checking within the scope of the compilation. The information passing between separately compiled procedures is limited to parameters and some set of common external names.

The online system allows program editing and submission. Sometimes it includes foreground or online execution and debugging.

The linkage editor (or loader) is used to connect separately compiled procedures together prior to execution time. It also connects library procedures into the resulting object deck.

The job control language processor is used to control the order of execution of the tools (eg. compiler and linkage editor) and specify the source and destination of all "files" (eg. procedure text, object deck, data etc.).

The system level routines are either linkedited into the object deck or are dynamically invoked at execution time by supervior calls (SVC's).

### 1.1.2 PROGRAM DEVELOPMENT PROCESS

Within the previously outlined system (section 1.1.1) a certain program development process emerges. This process is an approximation to the modular programming methodology which is restricted by the tools available.

The first step in the process is that of developing a complete problem statement. This is really a process which precedes the actual software development process and is the same in all environments.

The next step is the stepwise refinement process. This is carried out by the chief programmers [1]. It includes specification and documentation of procedure interfaces and behaviour, description of global and local data areas, and definition of access rights to procedures and levels of the program. The result of this step is a document (or documents) indicating the expected behaviour of the entire project. It may be stored online so that it may be accessed as needed by programmers during the development phase of the project.

Once the design phase is completed the programmer teams develop their portions of the project more or less independently. Within each team, development usually progresses from bottom to top, that is, the procedures which depend only on system routines, library procedures, and/or programming language features, are coded first and once they have been coded, tested, and debugged, procedures at the next level may be developed. An alternative is a top down approach where program stubs are used.

Each procedure is developed independently with reference to the design document for specification of interfaces and use of global data areas, etc. The procedure is compiled and tested independently of others and when it is thought to be correct, it is linkedited into the object deck for the entire project. Once this is done it provides part of a basis for the next level of development.

Sometimes, during the development phase, a design decision is found to be incorrect or incomplete. When this occurs, it is necessary to re-evaluate parts of the design phase and then modify the design document. After this has been done it is then the responsibility of each team to ensure that the procedures they have previously developed and are currently developing, conform to the new, modified design document. This is probably the most confusing and error prone part of the development process.

Once the development process reaches the top level of the design, an overall system test may be carried out and then the integration of the project into common use can begin.

## 1.2 PROBLEMS ENCOUNTERED IN CURRENT ENVIRONMENT

There are numerous problems in trying to use modular program development methodology in the current environment. These stem from the fact that the current environment was developed to support the development of a program as a single entity. Only when the need for independent development of separate portions of the same project was realised were some attempts made to adapt the environment to this purpose.

The major problem in the current environment is the lack of any control or verification of the interfaces between separately compiled parts of a program. The only method for controlling these interfaces is the design document which states what these interfaces should be. There is no automatic method of verifying that these proposed interfaces are the ones actually used. Thus it is quite possible, and often the cause of many problems, for one procedure to view an interface one way and another procedure to view it in some other way, causing incorrect intercommunication between these procedures. This error is not detected except after a painstaking search when some unexplainable bug is encountered after these two prodedures are used together.

Another area where lack of interface verification is a problem is that between the program and the operating system. The operating system has no a priori knowledge that a program is going to communicate with it correctly. It must, therefore, test at execution time whether the program's usage of the interface is correct if it is to make any test at all. Unfortunately, the number of ways in which the interface could be used incorrectly is large and the operating system cannot check for all of them. Also since this verification is done at execution time it is performed every time the interface is used and is thus very expensive.

When the design document is being developed there is no automatic method of discerning whether or not its specification of interfaces is consistent. It must, therfore, be thoroughly checked for consistency by its authors. This is a time consuming process and is definitely error prone.

When a design decision is reevaluated, and this leads to a modification of the specification of an interface or interfaces, there exists no mechanism which can ensure that the new interface specification is reflected throughout the project, both in existing, already tested procedures and ones under development. Sometimes the change in interface does not affect a procedure other than to change some data typing information; othertimes it may have far reaching effects and may demand redevelopment of a procedure.

A large portion of development time is spent in verifying the interfaces between procedures. It is not sufficient to test a procedure by itself but it must also be tested in conjunction with every other procedure with which it could interact. If there was a guarantee that a procedure used its interface correctly, that its only method of accessing external information was via that interface, and that any intercommunication with that procedure was via that interface, then a large portion of the testing could be eliminated since the definition of a procedure would be guaranteed consistent with its uses.

There is a requirement for users of the current environment to have a familiarity with many languages. In addition to the language in which the program is being written, the user is required to know the online system command language, the editor commands, the linkage editor commands and the debugging facility commands. This proliferation of languages causes time loss and confusion to the users.

Procedure testing and debugging is usually carried out at a lower level than that of the programming language. That is, most debugging packages allow only machine level interaction with the procedure to be debugged. This means that a user must understand the way in which the compiler he is using operates and represents the data structures of his program. If a debugging package is not used then either core dumps or traces must be used. Both of these produce a large amount of output which must then be searched for what is really desired. Again, the core dump and some traces are at the machine level and thus suffer from the same problems as the machine level debugging package.

A final problem in the current environment is the unit which can be developed independently. In most languages and compilers this unit is the procedure. This limits definition of interfaces to specification of procedure parameters and declaration and use of global variables. There exists no way of grouping together, into a single development unit, the data and the procedures which work on that data (i.e. an abstract type [14]) and specifying inter type rather than inter procedure interfaces.

# 2 A MODERN PROGRAM DEVELOPMENT ENVIRONMENT

## 2.1 ABSTRACT TYPES

The design of a system using stepwise refinement involves the definition of the behaviour of the system and the resources it provides to a user. The abstractions necessary to implement the system can then be specified by defining their behaviour and indicating what resources each provides. The aggregate of these abstractions defines an abstract machine which provides the resources desired by a user of the system [14]. In the next level of refinement the abstractions necesary to implement the abstractions of the previous level can be specified. Again, these can be aggregated into subsystems required to implement the abstract types of the previous abstract machine, and each of these subsystems is in fact an abstract machine at the lower level. Eventually the abstractions required to implement all of the remaining abstract machines will be available on the target machine, and the design will be complete.

The abstractions at each level are definitions of new abstract types with the operations defined upon the abstract type as part of the definition. These abstract types make up the modules of the modular programming methodology.

The abstraction is the design unit in which an abstract type is defined. It includes a definition of the abstract type's behaviour, a specification of the resources provided, and a specification of the abstract types required for this abstract type's implementation [8]. The specification of the resources provided and the abstract types required constitute the abstract type's interface with other abstract types in the design.

In the implementation of a program in a top down manner, an abstract type at one level is implemented in code for an abstract machine of the next lower level. The abstractions of the abstract machines at the lowest level are implemented in the code of the target machine.

The realization is the programming unit in which one abstract type of an abstract machine is realized. It consists of code, written for the underlying abstract machine, which realizes the resources specified in the abstraction according to the behaviour there defined and using the abstract types there specified.

An abstract type is thus two parts [12]: an abstraction which includes the module's interface with the external environ, and a definition of the abstract type's behaviour; and a realization.

When design and implementation are complete we have a collection of abstract types, each providing resources for some abstract types and each depending on resources provided by other abstract types.

## 2.2 CONSISTENCY CHECKING

As a program is being designed using the method indicated in section 2.1, a number of abstractions evolve. To verify that the design is correct it is necessary to prove these abstractions to be both complete and consistent.

Completeness means that all abstractions which have been referenced by any abstraction are defined in the system. Consistency means that the abstractions interact in a consistent manner, that is, reference only the resources provided by other abstractions and use these resources in a manner consistent with their definitions.

Proof of completeness is easily done by comparing references with the abstractions defined. The proof of consistency involves proving each abstract type's interface to be consistent with that of every other abstract type with which the first intercommunicates. This implies, for example, verifying that the type, number of actual parameters and types of the actual parameters in the reference match the type, number of formal parameters and types of the formal parameters in the resource definition. In addition the object referenced in the operation must be verified to be an instance of the abstract type for which the operation is defined.

During implementation, a realization is developed for each abstraction. It is necessary to prove this realization valid. To do this, it must be verified that the realization behaves as defined in the abstraction and that it uses the interface specified in the abstraction correctly.

Whenever a modification is made, be it to the abstraction or the realization of the abstract type, some reverification must be done. Depending on the change, the reverification may be localized or very general. If the change is to a realization, the reverification is localized and involves verification that the new realization is consistent with the abstraction for this abstract type. If the change is to an abstraction then it is necessary to verify that the realization for this module is still consistent with the new abstraction and that the interfaces of this abstract type and all abstract types with which it intercommunicates are still consistent.

## 2.3 SEPARATE COMPILATION

As was indicated in section 2.1, the implementation of an abstract type is perceived as a single problem during the implementation phase. This implies that the only concern during the implementation of an abstract type should be how it is implemented and not how it is to fit into the scheme of things for the entire project. The implementation proceeds with reference only to the abstraction defined during the design phase of project development. Thus implementation of an abstract type proceeds without knowledge of the abstract type from which it was abstracted [20]. This, in essence, is what modular programming is all about and what makes it a reasonable program development methodology.

Since the implementation of an abstract type proceeds independently of the implementation of other abstract types, so the verification of the implementation should proceed independently. The verification process includes a proof of the implementation's consistency with the behaviour and interface given in the abstraction.

The proof that the realization is consistent with the definition of the abstract type's behaviour is actually a program proof as defined in the literature [16]. Automatic program proving or a formal or informal manual proof may be used to verify this consistency. The compiler used to compile the realization may verify that the realization uses the interface specified in the abstraction correctly if the abstraction is compiled along with the module.

## 2.4 AUTOMATIC CONSISTENCY CHECKING

Separate compilation as described above (section 2.3) can provide automatic consistency checking on only two of the four areas described in section 2.2 (i.e. that abstractions are complete and consistent and that realizations behave as defined and use the interfaces correctly). The checks on completeness of abstractions and the checks of consistency of interfaces cannot be automatically done at compile time using separate compilation in current systems. The check of completeness can be done at link-edit time but the check of consistency of interfaces is done at execution time, if at all. This would require that code be produced at each point of access to every module to verify that the module has been accessed correctly.

This solution is expensive in three ways. First, it does not provide the check until the abstraction is tested with all other abstractions with which it interacts, at which time, if there are any errors, development must return to a point which was passed possibly months before. Secondly, of course, the interfaces are checked whenever the program is run causing extra execution time. Thirdly, the solution is incomplete, since only the interfaces actually used in the test are verified instead of testing all interfaces which may be used in production. If the operating system is developed in this same way, this solution could provide consistency checking of the interface between the modules in the project and the modules in the operating system.

A second solution providing automatic consistency checking is to eliminate separate compilation and force all abstract types in a project to be compiled simultaneously. In this way, complete consistency checking could be done since all information required by the compiler is available.

This solution is cost prohibitive. In a large project the number of abstract types to be compiled would be exceedingly large and compilation times would be long. Whenever a modification, however minor, were made to a abstract type, complete recompilation of the entire project would have to be done. In addition, to maintain complete consistency checking of the project to operating system interface at compile time would require a complete recompilation of the operating system and the project together which is essentially impossible.

What is desired is a system in which complete consistency checking can be done automatically at compile time, thus eliminating the errors and oversights common in manual checking, and providing all the information about consistency errors at the time that it is required. In addition, of course, the solution must allow separate compilation.

## 2.5 DESIGN GOALS OF DEMOS

The intent of the design of DEMOS is to provide a suitable software environment for the modular development of programs. The system will support the complete development process (section 2.1) from design through implementation and testing as well as the project's life in production. Complete consistency between all abstract types (section 2.2) will be enforced by the system, including those in the operating system, while allowing separate compilation of abstract types to be performed (section 2.3).

Inconsistencies will be detected and reported at compile time. The design phase will be conducted on-line so that any design inconsistencies can be detected at that time. Abstract types will be checked for consistency when they are entered during development. Whenever abstractions or realizations are modified, any resulting inconsistencies will be noted. At no time will an inconsistent abstract type be allowed to be utilized.

Data security will be provided by the enforcement of consistency in access to data by the compiler and be supplimented by access rights checking, by the execution environment, of access to instances of abstract types.

The system will be on-line to provide immediate feedback and will be integrated rather than a collection of tools.

## 2.6 REQUIREMENTS FOR THE DEMOS SOLUTION

The provision of the facility of automatic consistency checking at compile time while still allowing separate compilation, places certain requirements on the system. These requirements include: use of high-level language(s) only, retention of abstractions after compilation, provision of a mechanism to detect possible inconsistencies after a modification is made, and provision for the development of the operating system utilizing this system.

A high level language is necessary since only a compiler can do the required consistency checking. To allow a translator (for a language at any level) which does not or can not do the consistency checking to be used in this system would defeat the entire purpose of the system. The language(s) must not allow low level operations which could be used to defeat the consistency checking (e.g. unrestricted use of pointers etc.) for the same reason.

The abstraction provides valuable information to be used for consistency checking. It is created during the design phase and should not be included in the code for the realization but should be automatically used when the realization is compiled. After compilation the abstraction should not be discarded but retained so that other compilations may refer to it. When an abstract type is designed to interact with another abstract type, the abstractions are used to ensure consistency. After a modification is made, the abstractions may be used to reverify the consistency of the interfaces.

There must be a supervisory system which enforces consistency upon abstract types. It is this system which provides the correct abstractions to the compiler when compilation of a realization is being done. It also verifies completeness of the abstractions. A further requirement is that it be able to detect when inconsistencies are caused by modification of an abstract type and which abstract types are affected, and ensure that these inconsistencies are corrected before any execution is begun.

Finally, the operating system must have been developed within this system since project abstract types interact with operating system abstract types and the consistency of this interface must also be under control of the system.

# 3 LOGICAL ORGANIZATION OF DEMOS

## 3.1 MODULES IN DEMOS

The module concept in DEMOS is a language independent concept similar to that of the **class** concept in SIMULA [3], the **cluster** concept in CLU [14], [15], the **form** concept in Alphard [25], and similar constructs in other languages [18], [10], [24]. It embodies the data type abstraction discussed in [14] and section 2.1. The difference between the module in DEMOS and the **class** in SIMULA is in the ability to reference the individual parts which make up the entity. In SIMULA all names in the class may be referenced by any process which has access to any member of that class. In DEMOS, the module explicitly states which parts are visible (referable) outside the module and which are not.

The class definition in SIMULA and the module definition in DEMOS both are declarative in nature and provide the definition of a class of entities which could be created by anyone having access to the definition. In SIMULA, when a new member of the class is created (using the **new** operator), it appears to have all the facilities declared in the class definition but is distinct from any other member of that class. The same is true in DEMOS. When a module instance is created, it appears to have all the facilities defined in the module definition but is an entity different from any other instance of that module.

In DEMOS there are two things then, the module definition (hereafter called the module) and the instances of the module (hereafter called the module instances). The module serves as a definition of the behaviour and form of the module instances. The module consists of the two parts mentioned earlier (section 2.1), namely the abstraction and the realization. The module instance appears to include within itself the complete abstraction and realization and behaves accordingly, but is in fact only a data area (section 4.1.1) upon which operations specified in the module operate.

## 3.2 MODULE REPRESENTATION

A module is the development unit in the DEMOS system. The system provides for separate compilation of modules while retaining complete consistency checking. Modules in the system have interdependencies which force a new consistency check of a previously checked module to be performed whenever a module with which it interacts is modified, in order to maintain assurance of consistency. Since, in general, one module may interact with many other modules each of which may interact with others, and so on, the number of new consistency checks of other modules due to a change in one module may be large and an effort must be made to limit the number and complexity of these consistency checks.

To do this, DEMOS does not maintain a module as a single unit, but as a collection of pieces called components. The choice of what makes up a component is made with the need for limiting the complexity of the consistency checks in mind.

In addition to the components which make up a module, the system maintains a set of dependency relationships between module components, both within and among modules. A component A is said to be dependent upon a component B if:

a) A is another representation for the information in B (direct dependence)

b) there is an implicit mapping of entries in component B to entries in component A (form dependence)

c) there is an explicit mapping (i.e. a pointer, offset or index) from A to component B (Pointer dependence)

d) component A contains information taken from component B (information dependence)

Dependencies are used to determine when components must be subjected to new consistency checks. The rule used is: if component A is dependent upon component B and component B is modified, then component A must undergo a new consistency check.

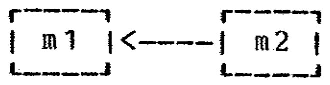


fig. 3.2.a Module Dependence

Logically, when a module m2 uses resources provided by a module m1, we have a dependence relationship such that m2 is dependent upon m1. This relationship can be viewed as shown in figure 3.2.a. Here a module m2 is dependent upon a module m1 (as shown by the arrow) since a change to module m1 must force a consistency check of module m2 to ensure consistency. In DEMOS, the logical dependence of m2 upon m1 is reflected by the actual dependence of some components of module m2 upon some components of module m1.

### 3.2.1 COMPONENTS

As was indicated in section 2.1, to a user a module is composed of two parts: an abstraction and a realization. These are two distinct entities developed at different times in the development process, the abstraction being developed during the design phase and the realization during the implementation phase.

In DEMOS the logical division of a module into two parts is reflected by a division of the module representation into two groups of components. The first group consists of components reflecting the definition of the module, i.e. the abstraction. This group also defines the interface of the module with other modules within the system. The second group reflects the particular realization of the defining abstaction. (see figure 3.2.1.a)

$$\begin{tikzcd} \boxed{\text{abstraction}} & \boxed{\text{abstraction}} \arrow[l] \\ \boxed{\text{realization}} \arrow[u] & \boxed{\text{realization}} \arrow[u] \\ m1 & m2 \end{tikzcd}$$

fig. 3.2.1.a Component Groups

Here the dependence of module m2 upon m1 is reflected by a dependence of the abstraction of m2 upon the abstrction of m1 and indicates that m2 uses resources provided by m1. The realization is dependent upon the abstraction since it is a realization of the module as defined by the abstraction.

The realization is separated from the abstraction since the realization has no effect upon the interface between this module and others. Any realization which realizes the module defined by the abstraction would suffice and a change to the realization does not require any new consistency checks of the interface between this module and others.

#### 3.2.1.1 ABSTRACTION COMPONENTS

The abstraction can be viewed as a definition part and an interface part. The definition part indicates the parameterization of the module and the machine upon which this module is to be implemented. The interface part indicates the interface that this module has with other modules.

The definition part is represented by a single component called the module specification (MS) which contains the parameterization and the name of the implementation machine for the module. A module may be parameterized so that different instances of the module may vary structurally but remain organizationally the same [25]. This parameterization includes parameters of the standard types within the system language and parameters of the type **type** (i.e. a module may be parameterized by a data type). This allows a module to be somewhat representation independent, at least in these parameters. For example: a single module "stack" could be defined, parameterized by an integer (the maximum stack depth) and a type (the type of elements in the stack). This one module could be used to define stacks of any depth and type including even a stack of stacks. The values of module parameters are not available until module instance creation time and thus the use of parameters cannot be consistency checked until that time.

The interface part is represented by two components, the uses specification (US) and the defines specification (DS).

The uses specification indicates the resources required by this module and the names of the module(s) which are expected to provide them. This component is dependent upon the module specification since the modules available to provide the resources needed belong to the machine listed in the module specification or are parameters of the module (as indicated by the module specification). The existence of the resources is known only to the module from which the resources are drawn. Thus the uses specification is dependent upon the defines specification of all modules from which this one draws resources.

The defines specification indicates the resources provided by the module to other modules within the system. For each resource it also gives complete typing information. Since the types may be modules which provide resources to this module, or types in the parameter list, this component is dependent upon the module specification of this module.

The components that make up the abstraction of a module are shown in 3.2.1.1.a. It can be seen that the dependence of one module's abstraction upon that of another is repre-

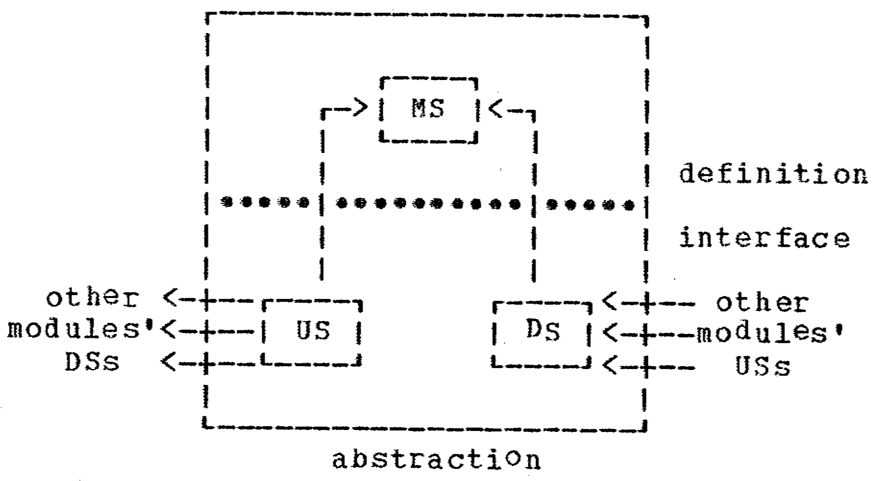


fig. 3.2.1.1.a Abstraction Components

sented by the dependence of that module's US upon the DS of the other.

The resources defined and used by a module are all operations upon the abstract type which is defined by the module. These are represented by procedures and operators. Data objects within the module are not provided as resources since this would cause a dependency of modules upon the representation (see section 3.2.1.2) of the module which is not part of the abstraction. If, in fact, access is desired to a data object from within another module, this access may be made via procedures defined for this purpose in the defining module. These procedures themselves are dependent upon the representation but the use of the procedure is not and thus the unwanted dependency between a module and the representation of another is avoided.

The three components of the abstraction are created by the designer of the module and thus originate in a human comprehendable (external) form. To facilitate consistency checking, an internal form would be preferred. DEMOS maintains an internal form of each of the three components of the abstraction.

The internal form of the module specification is called the module definition (MD). It consists of a pointer to a machine and a table containing one entry for each parameter in the parameter list. In addition the table contains one entry for each module available for use in implementation of this module. Each entry contains: the module or formal parameter name; a flag indicating whether the entry is for a module or a formal parameter; the type of the object described by the entry (modules have type **type**) in a coded form; and an address field which is either the offset within the parameter space to the parameter (if the entry is a parameter) or the module address within the system (if the entry is for a module which is not a parameter).

The internal form of the uses specification is called the import list (IL). A resource is said to be imported if it is provided by another module and used by this one. The import list contains one entry for each resource used by the module. Each entry contains: the resource name; the offset in the MD of the entry for the module providing the resource; and the index, into the export definition (ED see next paragraph) of the module providing the resource, of the entry for the resource used. This provides a link from the IL to the ED of the module providing the resource since the location of the ED can be found from the module address which is contained in the MD. Note however, that if the module providing the resource is a parameter, the ED index is not available until module instance creation time and differs with different module instances. Thus, for a module parameter, the ED index field is not used in the IL but resides in the parameter space.

The internal form of the defines specification is called the export definition (ED). It contains one entry for each resource defined by this module. Each entry contains: the resource name and the type of the resource in a coded form. The coded form includes an offset within the MD whenever a type defined by a module is used in the parameterization of the resource.

The module definition contains all the information contained in the module specification except for the machine name. Due to this fact, if the machine name is retained in the MD, the external form of the module specification can be regenerated from the internal form except for the layout (i.e. spacing etc.). If a standard layout is adopted, it is not necessary to maintain the external form at all, since, whenever the external form is desired, it can be regenerated from the internal one. Changes can be made by the user, directly to the internal form as well, (even though the user may think he is modifying the external form). Thus DEMOS maintains only the MD and an imaginary MS.

A similar argument can be used for the uses specification and the defines specification except for one thing. When the IL (or ED) is consistent with the MD, the pointer into the MD can be used to regenerate the module names referred to by the entry. However, if the MD is modified, the pointer is no longer valid, and there is no longer any way to regenerate the module name for this entry. To overcome this problem, another field is added to each of the IL and the ED. The field added to the IL contains the name of the module providing the resource. The field added to the ED contains an internal text form of the typing information of the resource. Now all the information required for regeneration of the US from the IL and the DS from the ED is avaILable for listing, modification or consistency checking purposes, and so DEMOS maintains only the IL and ED and an immaginary US and DS.

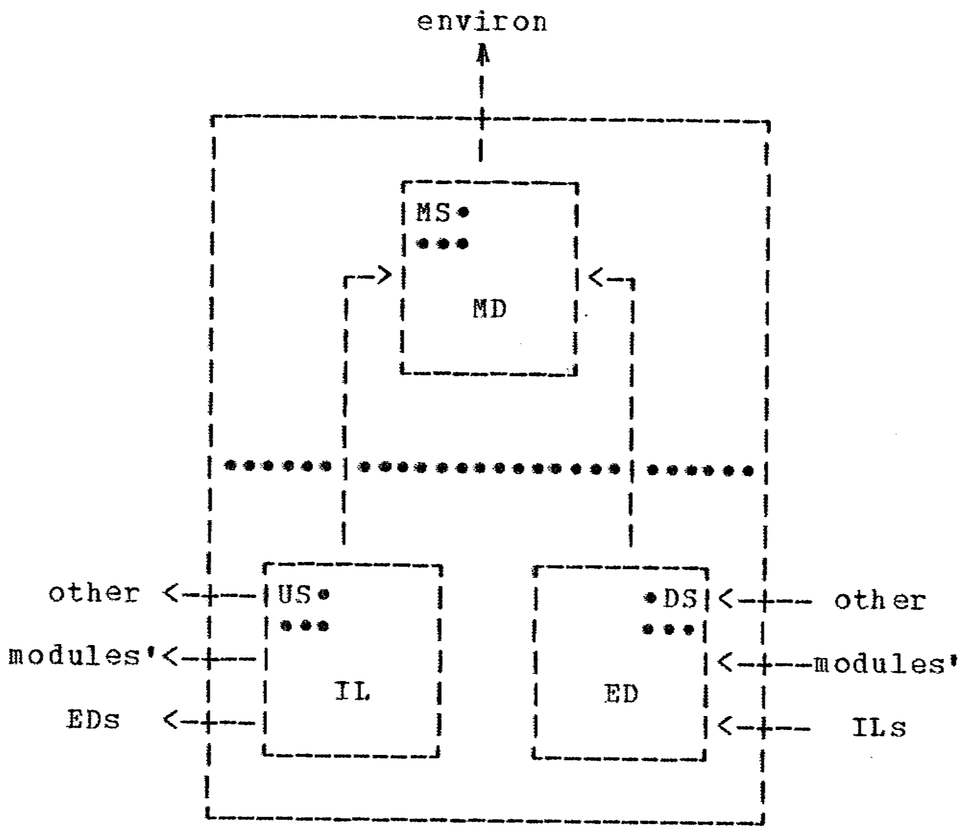


fig. 3.2.1.1.b Abstraction Representation

The final form of the abstraction part of the module representation is shown in figure 3.2.1.1.b. Here the IL and the ED are dependent upon the MD by pointer dependence since they contain offsets into the MD. The IL is dependent upon the EDs of other modules which supply resources to this module by pointer dependence since it contains indices into these EDs. The MD has a dependence as well since it is dependent upon the source of the module names and addresses by information dependence. This source is termed the environ of the module and is discussed in section 3.4.

#### 3.2.1.2 REALIZATION COMPONENTS

The realization of a module can be viewed as two parts, a representation part and an implementation part [12], [5]. The representation part indicates the concrete representation of the abstract data type defined by the module and the implementation part indicates the implementation of the operations defined upon that type.

The representation part is represented by a single component called the module representation (MR). The module representation contains declarations of all variables (fields) which make up the concrete static representation of the abstract type. This representation defines the instantiation of the module (i.e. the module instance or data area). The typing information includes references to modules listed in the module definition component of the abstraction part of the module.

The implementation part is represented by a single component called the module implementation (MI). The module implementation contains the source code for operations defined by or local to the module, and the code for creation and disposal of the module instances. The source code makes reference to the resource declarations given in the DS of this and other modules, uses only resources listed in the US and references modules of the environ indicated by the MS and the variables declared in the MR.

The realization part is thus represented as shown in figure 3.2.1.2.a.

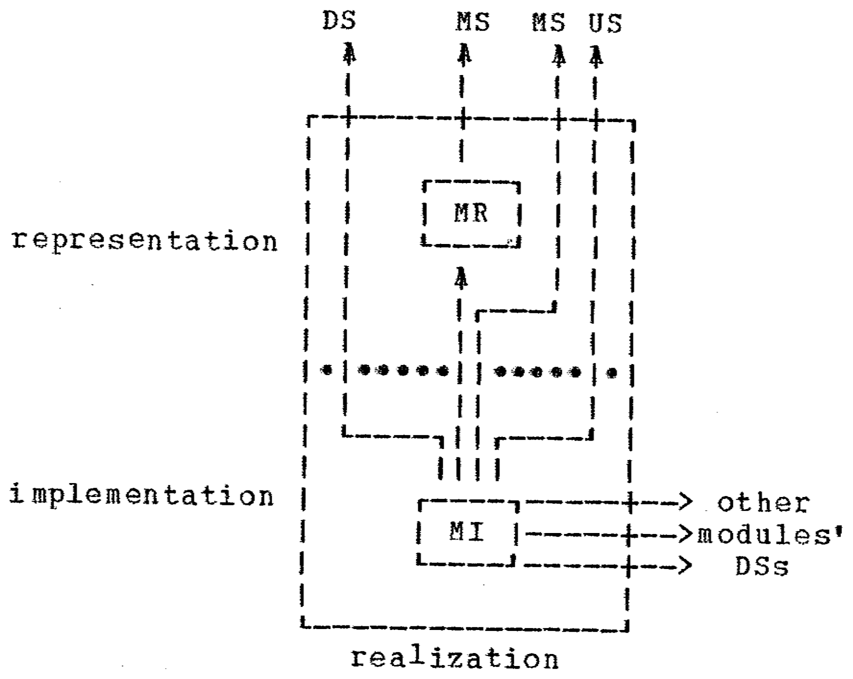


fig. 3.2.1.2.a Realization Components

The two components of the realization were created by the implementor in a human comprehendable (external) form. Execution speed can be improved if internal forms of the components are maintained, since this would obviate the necessity of scanning the MR and interpreting the MI during execution. To facilitate this improvement, DEMOS maintains some internal components for the realization part of the module definition.

The internal form of the module representation is called the representation dictionary (RD). The representation dictionary contains one entry for each field (variable) in the representation of the abstract data type. Each entry contains the field name, the type of the field (encoded in an internal form) and the address of the field within the data area. The internal form of the type includes an offset into the MD or standard type table to entries for the names of the modules which define the types of the fields.

The RD contains all of the information from the MR as long as it is consistent with the MD. However, this consistency may be violated if the MD is modified and, in this event, it is impossible to determine the types of the fields directly from the coded type in the RD (since this contains offsets into the MD which are now invalid). To enable the maintenance of only the RD and not the MR, a new field must be added to the RD which contains the type of the field in an internal text form. This now allows the regeneration of the MR from the RD and thus eliminates the need to maintain both the MR and the RD.

The internal form of the MI is more involved. Clearly the most important component is the object code (OC) created from the MI. This object code requires the addresses of all resources which it utilizes including those provided by other modules. If these addresses were coded directly into the OC, they would force the OC to be dependent upon the OCs of other modules by pointer dependence. This is undesirable since this would make the implementation of one module dependent upon the implementation of another which violates the desire that modules be implementation independent.

To overcome this, a new component is added to the implementation part of the realization. This component, called the entry map (EM), contains the offsets of the resources defined by this module within the object code component. This enables a reference to this table by the system at execution time when a call is made to a resource in this module, in order to get the address of the resource.

This, however, has two drawbacks. Firstly, this reference must be done at each use of the resource even if its address has not changed since the last use and, secondly, requires system intervention at execution time to resolve the reference.

To remove these drawbacks, another component is maintained as part of the implementation. This component, called the use map (UM), contains one entry for each external name referenced. This entry contains the offset within the OC of the providing module of the resource referenced. These offsets are loaded from the EMs of the providing modules. All references to external resources from the OC are made indirectly via the UM. As long as the UM is consistent with the EM of a providing module, the reference to the resource can be made without system intervention. If the UM is not consistent with the EM, the new offsets in the EM must be loaded into the UM before execution may proceed.

Since there is one entry for each resource referenced by the module implementation in the UM and there is one entry for each resource referable in the IL, there is an entry in the IL for each entry in the UM. In fact, in a fully implemented module, there should be a one-to-one correspondence of entries in these two components. The UM could even be considered as part of the IL except that it is dependent upon implementational not abstraction considerations. Due to this correspondence, the UM is maintained only as a table of offsets organized so that there is a correspondence between the entries in it and the entries in the IL such that the first entry in the UM is the offset of the resource described by the first entry in the IL, etc.

A similar argument can be used for the structure of the EM resulting in the EM being simply a table of offsets which has a one-to-one correspondence to ED entries such that the first entry in the EM is the offset for the resource described in the first entry of the ED, etc.

It is now possible to do an efficient job of loading the UM when necessary. If the IL is consistent with all EDs upon which it depends, it contains indices into the EDs (and hence EMs) of the providing modules and its entries correspond one-to-one with the entries in the UM to be loaded. Thus a series of simple index and copy operations can be used to load the UM from the required EMs.

A final component is added to the implementation part of the realization of a module. This component, called the symbol table (ST), is added for consistency checking of separately compiled procedures (section 3.2.1.3). The symbol table contains one entry for each name declared in the MI. Each entry contains the name, the encoded form of the type, the scope of the name and its offset or value. This component is another internal form of the MI.

The OC and the ST together do not give all the information contained in the MI since the translation of source language statements into machine language instructions does not have a well defined reciprocal operation. Thus, the MI must be maintained in order to have a human readable form, unlike the cases for the MS, DS, US, and MR.

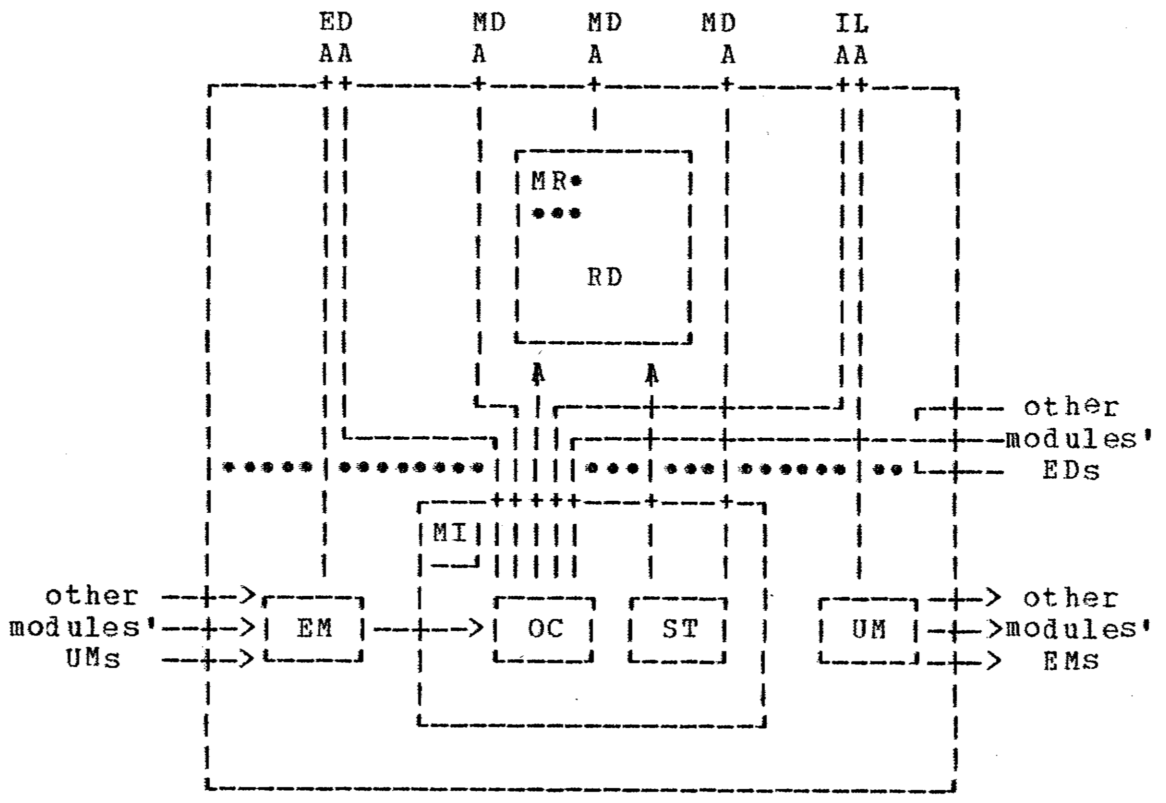


fig. 3.2.1.2.b Realization Representation

The final representation of the module realization is shown in figure 3.2.1.2.b. Here the RD is dependent upon the MD by pointer dependence since it contains indices into the MD (i.e. to module names defining the types of the fields). The MR does not actually exist but is an imaginary component.

The OC and ST are directly dependent upon the MI since they are internal forms of the MI. The EM is dependent upon the OC by pointer dependence, since it contains offsets into the OC, and upon the ED by form dependence since its form is defined by the form of the ED. The UM is dependent upon the IL by form dependence and upon the EMs of other modules by information dependence. The OC is dependent upon the ED, RD, MD, IL and EDs of other modules by information dependence since information from these components is implicitly encoded in the OC. The ST is dependent upon the MD by pointer dependence since it contains indices into the MD.

#### 3.2.1.3 SEPARATE PROCEDURES

The module concept provides data type abstraction; however, this is probably not sufficient for program development [14]. Additionally a functional abstraction is needed. Functional abstraction is represented in DEMOS by procedures within a module. In order that development of a functional abstraction may proceed separately from that of the rest of the module, DEMOS provides for a special type of procedure, termed a separate procedure.

A separate procedure is no different functionally from any other procedure. Its difference lies in the manner in which consistency checking is done. A separate procedure is consistency checked at a time potentially different from that of the rest of the procedures in the module.

To enable this independence, a separate procedure is maintained as a separate entity within the module representation. Since its environ is like that of any other procedure within the same module, it is essentially another part of the implementation part of the module. The separate procedure is represented by a component called the procedure implementation (PI). The procedure implementation is the source code of the procedure body. It makes reference to the MS, DS and US of the abstraction part of this module, the MR and MI of the realization of the module and the DSs of modules supplying resources used by this procedure. A separate procedure is represented as shown in figure 3.2.1.3.a.

```
MS DS US MI MR
 ▲  ▲   ▲ ▲  ▲
 |  └┐  |┌┘  |
 └──┐|  ||┌──┘
    ||  |||
   ┌─────┐───> other
   | PI  |───>modules'
   └─────┘───>  DSs
```

fig. 3.2.1.3.a Separate Procedure Components

It should be noted that the procedure header (declaration) resides in the MI and not in the PI, that is, the procedure is declared in the MI and implemented in the PI. This is to enable references to the separate procedure by the MI to be consistency checked without being forced to compile the PI.

As expected, an internal form of the PI is maintained by DEMOS. This is similar to that of the MI and consists of two components, the procedure object code (PO) and the procedure symbol table (PS), which are organized in the same manner as their corresponding components in the implementation.

A functional abstraction may be made at any level, and, in fact, may be made from a separate procedure. In this case one separate procedure may be dependent upon another, higher level separate procedure in which its procedure header resides. Of course it is also dependent upon all separate procedures upon which its encompassing separate procedure depends as well as upon the MI.

The entry point of a defined resource must reside in the OC of the module and thus a separate prodecure may not implement a defined resource directly. This is due to the fact that the entries in the UM of a module using the resource are assumed to point into the OC of the defining module and this would not be true for a separate procedure. This does not pose any restriction since a defined resource could be implemented by a dummy procedure in the MI which simply invokes the separate procedure.

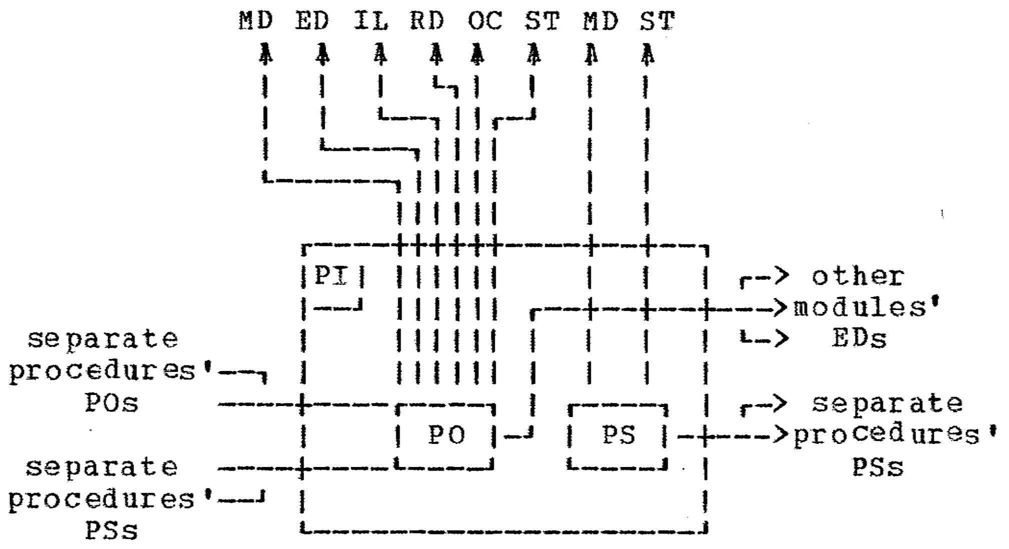


fig. 3.2.1.3.b Separate Procedure Representation

The final form of a separate procedure is shown in figure 3.2.1.3.b. Here the PO and PS are directly dependent upon the PI. The PO is dependent upon the MD, ED, IL, RD, OC and ST of the module, POs and PIs of other separate procedures in which it is imbedded within the module, and EDs of modules which supply resources utilized by this separate procedure, since the PO contains information from these sources encoded within it. Since the PS contains indices into the MD and ST of the module and PSs of separate procedures in which this one is imbedded, the PS is pointer dependent upon these components.

#### 3.2.1.4 MODULE INSTANCES

A module instance is one instantiation of the abstract type defined by a module. It thus incorporates the definition of the module as well as a data area in which the values of the fields of this instance are retained. Thus a module instance could be represented by a copy of the components that represent the module with the addition of a data area unique to this instance.

This representation would be inefficient, however, since it implies duplication of the components common to all instances of the same module. In addition, whenever a change is made to the module, all instances would have to be updated to reflect this change. A far more efficient representation is realized if only one copy of the module is maintained and each module instance is represented by only a data area which contains values unique to this instance.

Within DEMOS a module instance is represented by a data area (section 4.1.1). This data area (DA) is defined by the representation of the module (i.e. the MR) and is thus dependent upon the RD by form dependence (i.e. the RD defines the form of the DA). A data area is solely dependent upon the RD (see figure 3.2.1.4.a) which means that only a change to the MR will make a data area inconsistent.

$$\begin{tikzcd}
RD \\
\boxed{DA} \arrow[u]
\end{tikzcd}$$

fig. 3.2.1.4.a Data Area Representation

### 3.2.2 COMPONENT DEPENDENCIES

The components (figure 3.2.2.a) which make up the module representation discussed in section 3.2.1 have dependencies between them which arise from the inherent dependence of the abstract types upon other abstract types in terms of which the first are defined; the dependence of the realization of the abstract type upon the abstraction which defines the abstract type; and finally the desire to improve efficiency by maintaining internal forms of some module components. These dependencies indicate the components whose modification would require the reverification of consistency of a component.

| ACRONYM | COMPONENT | SECTION | TYPE |
|---|---|---|---|
| MS | Module Specification | 3.2.1.1 | U |
| DS | Defines Specification | 3.2.1.1 | U |
| US | Uses Specification | 3.2.1.1 | U |
| MD | Module Definition | 3.2.1.1 | S |
| ED | Export Definition | 3.2.1.1 | S |
| IL | Import List | 3.2.1.1 | S |
| MR | Module Representation | 3.2.1.2 | U |
| MI | Module Implementation | 3.2.1.2 | U |
| RD | Representation Dictionary | 3.2.1.2 | S |
| EM | Entry Map | 3.2.1.2 | S |
| OC | Object Code | 3.2.1.2 | S |
| ST | Symbol Table | 3.2.1.2 | S |
| UM | Use Map | 3.2.1.2 | S |
| PI | Procedure Implementation | 3.2.1.3 | U |
| PO | Procedure Object | 3.2.1.3 | S |
| PS | Procedure Source | 3.2.1.3 | S |
| DA | Data Area | 3.2.1.4 | S |

TYPE: U - User component
S - System component

fig. 3.2.2.a Module Component Summary

In any system under development, the number of modifications to components would be large and thus DEMOS will spend a large portion of its time performing consistency checks. It is important to minimize the number of components that must be interrogated to determine if a component need be consistency checked. To this end, the number of dependence relationships maintained in DEMOS is reduced to the minimum number needed to maintain complete consistency.

The first method used to reduce the number of dependencies that need to be maintained is due to noting that if there is a sequence of dependencies A->B->C and an additional dependence A->C, the dependence of A on C is not required since the discovery of C's modification will cause B to be consistency checked (modifying B) and this modification will lead to the consistency checking of A. This removes the need for the maintenance of the dependencies of: the EM upon the ED; the OC upon the MD and the EDS of other modules; the PO on the MD, ED, EDs of other modules, IL, RD and, if the separate procedure is local to another separate procedure, the OC, ST and POs and PSs of separate procedures other than that in which it is imbedded.

A second set of dependencies can be removed by noting that whenever the OC is regenerated (by recompiling the MI), the ST is recreated and vice versa. This implies that any dependencies that the ST has in common with the OC are not needed since the consistency checking of the ST will occur at the same time as that of the OC anyway. This removes the dependence of the ST upon the MD. The same argument works for the PS and the PO of separate procedures, removing the need for the dependence of the PS on the MD, ST and PSs of other separate procedures.

The dependence of the POs of separate procedures local to the module upon the ST can be removed by noting that, since the OC and ST are modified at the same time, if the ST is modified, the consistency check indicated by the PO->ST dependence will be forced by the PO->OC dependence. The same is true for the dependence of the PO of a separate procedure local to another separate procedure upon the PS of that procedure.

Lastly, the dependence of the EM upon the OC can be removed since, by necessity, the EM must be recreated at the same time as the OC since that is the only time that the information necessary to create the EM is available.

This leaves the dependency relationships shown in figure 3.2.2.b to be maintained by the system. The direct dependencies are not shown by arrows but by the inclusion of a component in the same box as the component it is directly dependent upon.

## 3.3 CONSISTENCY OF MODULE COMPONENTS

### 3.3.1 CONSISTENCY AND COMPILATION

The three areas of consistency mentioned in section 2.2 must be verified for all modules within the system to be

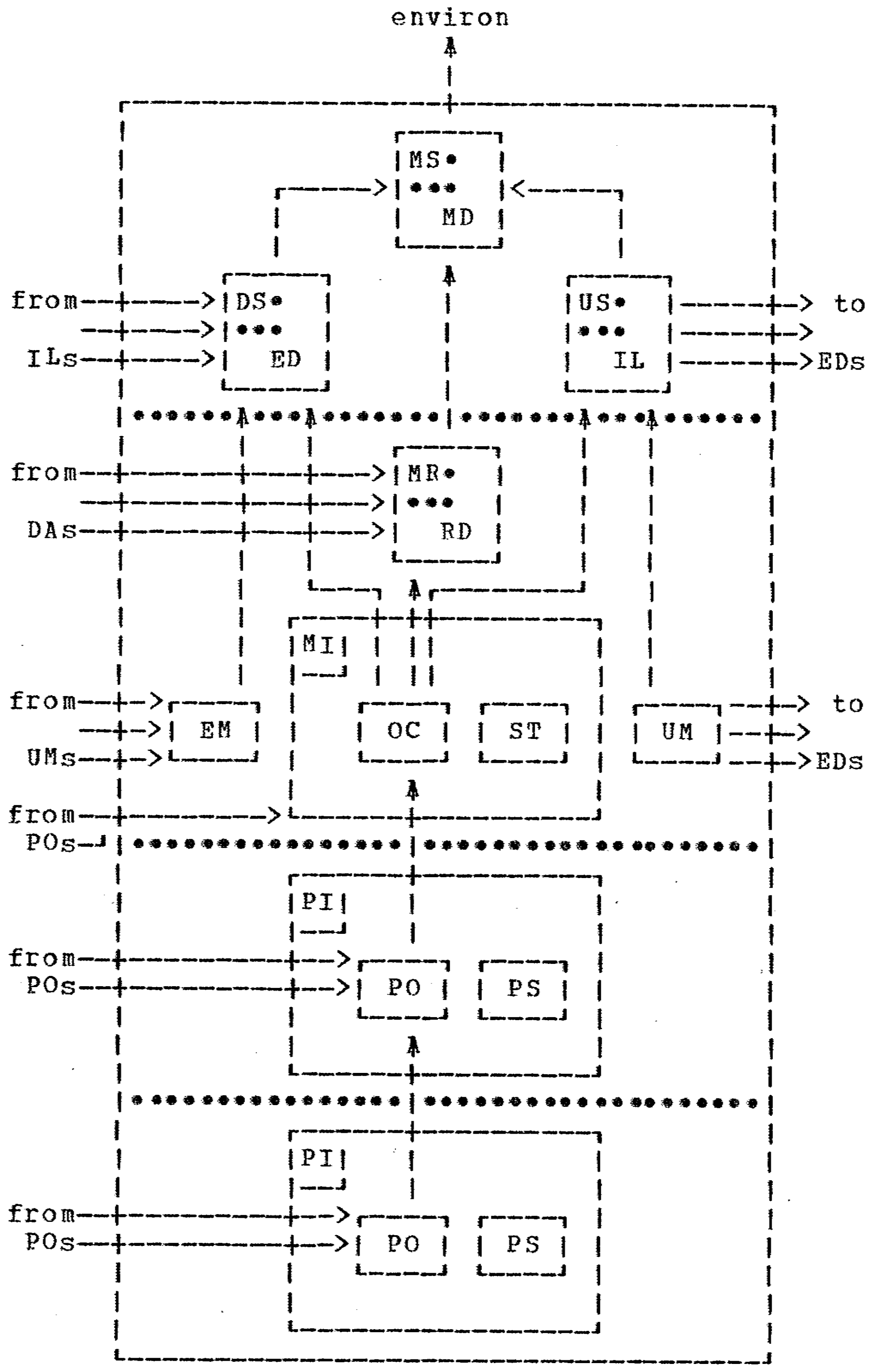


fig. 3.2.2.b Component Dependencies Maintained by DEMOS

certain that the system will function correctly. The first of these areas is completeness of the abstractions; that is, when an abstraction states that the module uses a resource provided by a second module, that second module exists provides the named resource, and secondly, that all external resources used by a realization are stated in the abstraction. The second area is consistency of utilization and provision of resources by the realizations; that is, that all resources declared as provided by the abstraction are realized in the realization consistent with their definition in the abstraction, and secondly, that the usage of external resources by the realization is consistent with the resources' definitions in the abstractions of the modules which provide the resources. The last area is consistency of the behaviour of the resources provided by the module with the definition of that behaviour in the abstraction.

A compiler is capable of verifying the consistency of the first two areas listed above if it has access to the MS, US, DS, MR, MI and PIs of all modules concerned. At the current time, compilers are not able to perform the consistency check upon the third area, which amounts to automatic program proving, but progress is being made in that area [11].

With the introduction of system maintained components into the module representation, a new type of consistency must be maintained. This is consistency of the information encoded in these components (the MD, ED, IL, RD, EM, OC, ST, UM, PO and PS) with the sources of that information. This differs from consistency in the user created components (i.e. the MS, US, DS, MR, MI and PI) in that inconsistency of system components implies that they were created from obsolete versions of user components while inconsistency of user components indicates a flaw in the design or implementation of one or more modules.

The system components can all be generated by a suitable compiler using the user components as sources. This compilation and generation process does consistency checking of the source (user) components as well as creating new, consistent system components. Since the system maintains internal forms of user entered components, the compiler need not use the user components as sources at all times but may use the internal forms of them as long as they are consistent with their external forms. This reduces the time required to generate new, consistent system components when old ones are inconsistent.

Since it is desirable to reduce the number (and duration) of the compilations needed to maintain and verify consistency, the compiler is broken into parts capable of compiling single user components using system components as additional sources. This allows regeneration of an inconsistent system component without recompiling all other module components.

There is effectively a "compiler" for each type of user component, that is, an MS compiler, a US compiler, a DS compiler, an MR compiler and an MI compiler. Since the PI is similar to the MI the same compiler can be used for the PI as the MI. Since the MS, US, DS and MR are not maintained concretely within the system, their compilers are really just table lookup and update routines invoked directly when the user wishes to modify one of these components. The MI compiler is the only real compiler within the system.

### 3.3.2 CONSISTENCY AND MODIFICATION

Whenever a component is modified, a check must be made to see if this component and all components dependent upon it are still consistent within the system. If there is an inconsistency, some action must be taken to reconcile it.

The dependencies recognized by DEMOS are those discussed in section 3.2.2. The maintenance of consistency of components related by these dependencies is sufficient to ensure consistency of all modules within the system. All the dependencies maintained are of system components upon either the user component of which they are an internal form or upon other system components. A system component is inconsistent whenever any component upon which it depends is modified. To restore the consistency, the system component is regenerated by "compilation" of the user component of which it is the internal form.

This compilation can result in a new, consistent version of the system component or will fail indicating that there is an inconsistency between two or more user components one of which is the one being recompiled, and the others being some of the indirect sources for the compilation.

Whenever a system component is discovered to be inconsistent, an attempt is made to regenerate the component. This attempt may either successfully complete yielding a consistent component, or fail indicating an inconsistency between user components. This second case requires intervention by a user to reconcile the inconsistency before a consistent system component can be generated.

Some components do not fit this scheme. The EM, ST and PS have no dependencies but are regenerated in consistent form as a byproduct of the regeneration of the OC and PO. The UM has no user component upon which it is dependent. When the UM is found inconsistent, it is regenerated by a special routine which accesses the IL and EMs of other modules. Regeneration of the UM cannot fail if the IL and the EMs are consistent and thus cannot indicate any inherent inconsistency betweem user components.

The inconsistency of the DA with the MR is an unreconcilable inconsistency since it implies that the DA no longer has the form defined by the MR and thus the data within it is incomprehensible. Recreation of a consistent DA would have to be done by a user by reinvoking the operations which created the DA using the new module representation and the old data.

When the MD is discovered inconsistent within the system, its regeneration will either be successful, indicating the MS is consistent, or will fail, indicating an incompleteness of the abstractions since the MS references a module which does not exist within its environ.

When the ED is discovered inconsistent with the MD, its regeneration is either successful, indicating the DS is consistent with the MS, or will fail, indicating the DS refers to some module not listed in the MS.

When the IL is discovered inconsistent with the MD or EDs of other modules, its regeneration is either successful or fails, indicating either an inconsistency between the US and MS (i.e. the US refers to a module not listed in the MS) or an inconsistency between the IL and the DS of some other module (i.e. an incompleteness of abstraction since the IL refers to a resource not provided by the module named).

When the RD is discovered inconsistent with the MD, its regeneration is either successful or fails, indicating an inconsistency between the MR and the MS (i.e. that the MR refers to a module not listed in the MS).

When the OC is discovered inconsistent with either the ED, RD, or IL, its regeneration is either successful, in which case new, consistent versions of the EM and ST will also be produced; or will fail, indicating an inconsistency between the MI and the MS, DS, MR, US and/or the DSs of other modules. These inconsistencies are inconsistencies of the realization of a module with the abstractions defining the module or those defining modules utilized by this one.

When the UM is discovered inconsistent with the IL or the EMs of other modules, its regeneration is always successful since its regeneration simply involves copying of information from the EMs in the order defined by the IL.

When the PO is found inconsistent with the OC or the PO of another separate procedure, its regeneration will either be successful, in which case a new, consistent version of the PS will also be generated, or will fail, indicating an inconsistency between the PI and the MS, DS, US, MR, MI, PO of another separate procedure and/or the DSs of other modules.

The regeneration of system components found inconsistent due to modification of a component upon which DEMOS knows the component is dependent, results in consistency checking of the user components in the two areas discussed in section 3.3.1 and thus verification of consistency of the modules in the system.

### 3.3.3 COMPILATION FOR CONSISTENCY CHECKING

The timing of the compilations to perform consistency checking is important. Along with the desire to report a user component's inconsistency as soon as possible, there is the requirement of informing the correct user, that is, the user with the knowledge and responsibility to reconcile the inconsistency. Also, since compilation requires time and will cause a real-time delay to the user who must wait for it to be done, it is important to cause the appropriate user this delay. The solution derives from dividing the dependencies into two categories and performing the requisite consistency checks for these two categories at different times.

### 3.3.3.1 CATEGORIES OF DEPENDENCE

The scheme used is based upon the assumption that different groups of people develop different modules and that within one group, two subgroups exist, one developing the abstraction and one the realization. Under this assumption, a change to an abstraction should cause reporting to the changer of any inconsistencies the change causes within the abstraction or between it and the abstractions of modules upon which this one depends. Likewise a change to the realization should cause reporting to the changer of any inconsistencies within the realization or between the realization and the abstraction it implements.

Two sets of inconsistencies are not covered by this. These are the inconsistency of the realization with the abstraction defining it after a change to the abstraction, and the inconsistency of an abstraction with the abstraction of a module it is dependent upon when the providing module's abstraction is modified. In both of these cases, the person making the change has neither the responsibility nor the expertise to understand messages concerning the inconsistency nor make the modifications required to reconcile them.

Consistency checking is thus divided into two categories: immediate and deferred. Components whose dependencies fall into the immediate category are consistency checked whenever the module upon which they have this dependence is modified. Components whose dependencies are in the deferred category have the implied consistency checks deferred until a later time which, however, must preceed utilization of the resources defined by this module.

Components which fall into the immediate category are those whose regeneration will cause consistency checking of user components which may result in reporting of inconsistencies which are the responsibility of the changer to reconcile. These components are all components which are internal forms of a user component with their dependence upon that user component, and all dependencies which are entirely within the abstraction or realization.

The modification of the MS causes the MD to be regenerated, which causes the ED and IL to be regenerated. When the DS is modified, the ED is regenerated, as is the IL when the US is modified. In the realization, whenever the MR is modified, the RD is regenerated causing the regeneration of the OC (and hence the ST and EM). Modification of the MI causes the regeneration of the OC (ST and EM). In a separate procedure, the modification of the PI causes the regeneration of the PO (and hence the PS).

Components with dependencies in the deferred category are those with dependencies which cross abstraction-realization, separate procedure or module boundaries. This includes the dependencies of the MD upon the environ; the IL upon the EDs of other modules; the OC upon the ED, RD, and IL; the UM upon the IL and EMs of providing modules; the POs on the OC or POs of other separate procedures; and the DAs upon the RD.

A couple of exceptions to the rule for immediate consistency checking may be made to allow the user more flexibility. Normally, a change to the MS would imply a change to the DS, US or both. In this case, if the MS were changed first, the unchanged versions of the DS and US would be consistency checked with the obvious inconsistencies noted. Likewise, if the DS or US were modified first, a spurious consistency check would be made. This is wasted effort since the user is well aware of these inconsistencies. In addition, a user may be making a number of changes to the MI or PI of a separate procedure but be unable to complete them all in one session. With only some of the changes made, recompilation of the MI (or PI) would be a waste.

To resolve this problem an option would be available to allow a user, when making a change to the MS, MI or PI, to specify that the consistency check be deferred. In this case, modification of the MS would not automatically cause the immediate regeneration of the ED and IL (as would normally be the case) and the modification of the MI (or PI) would not automatically cause the immediate regeneration of the OC (PO) and associated components. The consistency checks involved would instead be handled as deferred consistency checks by DEMOS.

### 3.3.3.2 DEFERRED CONSISTENCY CHECKS

Consistency checks which have been deferred must still be performed at some time prior to the use of the component in execution or another component's regeneration. In DEMOS, deferred consistency checking is performed just prior to the beginning of execution. The dependencies which could be involved in deferred consistency checking are shown in figure 3.3.3.2.a.

When execution of a resource from a module is initiated by a user (section 3.4.2.2), the system must determine if any deferred consistency checking must be performed before execution may begin. If some is necessary, it informs the user and allows the user to cancel the execution or to allow the system to proceed with the checking.

To determine which deferred consistency checks must be performed, the system follows the network of modules upon which this one depends directly or indirectly and places the modules encountered into sets, one set for each diferrent path length from the root module (i.e. the module in which the execution is to begin). Each module is placed only into the set for its maximum path length from the root. In doing so it also determines if the modules referred to exist in the specified environ (section 3.4).

When this operation terminates (as it must since ultimately, all modules depend upon the hardware) the system then begins with the set containing the modules at maximum disance from the root and tests the components of these

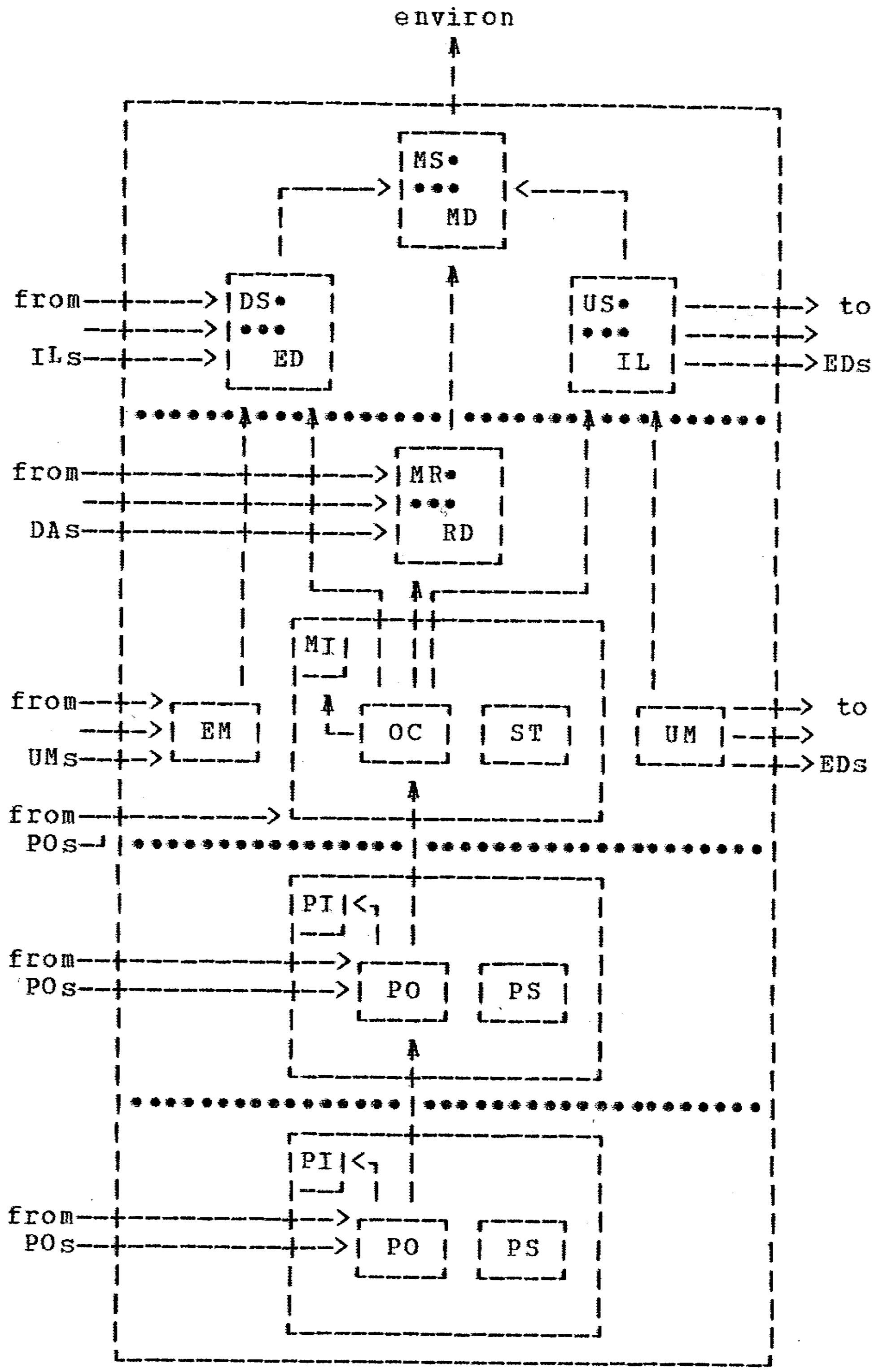


fig. 3.3.3.2.a Deferred Consistency Checks

modules to determine if any deferred consistency checks must be performed. If there are any, the checks are performed. When all the checking in this set is complete, the system goes on to the next set again testing for deferred consistency checks and, if any are found, does the checking and so on to the next set until the root module has been tested.

Within a module, the test for deferred consistency checks proceeds down from the MD through the ED, IL, RD, OC, UM to the POs in turn. Whenever a deferred consistency check is required, the check is performed before going to the next component.

To enable the system to determine when a consistency check has been deferred, components are date stamped with the date of their last modification. Whenever a user component is modified or a system component is regenerated, the date of last modification is updated. This requires the system to make only a simple test of the date stamps of a component and the components with which it may have a deferred consistency check to determine if a deferred consistency check is necessary. A deferred consistency check is necessary if the date stamp of the dependent component is prior to the date stamp of the component upon which it depends.

When deferred consistency checking is performed, the regeneration of the dependent system component may be successful, or unsuccessful due to an inconsistency between two or more user components. When the check is unsuccessful, the user is informed and further checks and the execution are aborted. To prevent future, unnecessary deferred consistency checks of a component which has already been unsuccessfully consistency checked, the system maintains a flag in each component which is set when a consistency check is successful and reset when it is unsuccessful. In addition, even if an unsuccessful regeneration of a component occurs, the modification date is set. When a component is tested for deferred consistency checking, the date stamps are tested as usual. If they indicate that a deferred check is necessary, the check is done regardless of the state of the flag. However, if the dates indicate that no deferred check need be done, the system tests the flag, and, if it is reset, the component is treated as if it had undergone an unsuccessful consistency check. This eliminates unnecessary compilations of components which are already known to be inconsistent.

## 3.4 ABSTRACT MACHINES

Modules in DEMOS are implemented upon abstract machines (or simply machines). Where modules are the development unit, machines are the organizational unit. Each module is a module of one and only one machine and gives to the machine the capabilities of the data type defined by the module and the operations upon it. In this way a module is like a hardware block within a CPU (eg. the integer arithmetic hardware) and a machine is like a hardware machine.

A machine has the ability to execute operations in its repertoire upon data objects located within its store. In terms of modules, the operations are the resources provided by the modules making up the machine, and the data objects are instances of those modules. These operations may be initiated and sequenced by a user attached to the machine or by a super machine (machine at a higher level running on this machine) acting as an automated user.

When a system is being developed, a new machine is created to provide users with the capabilities desired in the system. This machine is developed in terms of its composite modules. Each module making up the machine is to be implemented on some other machine (or subsystem) which is called the environ for the module. This machine may be the same for all modules which comprise the current machine, or there may be different machines used to implement different modules. At any time, some of the machines needed to implement modules may be previously defined in other systems or may be the actual hardware machine. If this is the case for all modules in a machine, no new machines need be created. However, if, for some modules, new machines are needed, they can be created as described above.

When a user is attached to a machine, he uses the machine language for that machine to perform his desired tasks. This machine language enables the user to utilize the facilities of the modules provided upon the machine. The language is, in fact, the implementation language extended by the abstract data types which are defined on the machine and the operations upon those types. The unextended language is the same for all machines in the system.

When a user creates an instance of an abstract type during his work on a machine, that instance is created in the address space (see section 4.2.1) which is the store for the machine. The instance will be created as either a dynamic or static area (i.e. as an address space or not) depending on whether the representation of the module involves dynamicly sized data or only data of static size. The user may, upon creation of an instance, supply access rights information concerning this instance, to limit other users access beyond the limitations of the machine that the instance is created within.

An address space in DEMOS is a region of storage within which dynamic allocation may be done and for which dynamic address translation is performed. A more detailed discussion is given in section 4.2.1.

Any operation that may be done by a user on a machine may also be done by a super machine running on the machine. This is the way modules operate upon the abstract types in terms of which they are defined. The differences between this mode and that of a user using the machine are, firstly, that instances created by a super machine are not necessarily created within the address space of this machine but are usually allocated as subpaces of the user created data area upon which the operations ultimately causing this creation were initiated by the user (see section 3.4.3). Secondly, instructions from the super machine which are executed by the machine are not necessarily in the implementation language (which looks to a user as the machine language for the machine) but are usually in the actual machine language of the target machine.

### 3.4.1 MACHINE REPRESENTATION

A machine consists of four basic units. These are the control unit, the operations unit, the memory management unit, and the memory unit. These basic units also exist for a DEMOS machine. The control unit is an interpreter for the implementation language. The operations unit is represented by the collection of modules which makes up the machine. The memory management unit is a module which defines the address management operations on the machine. The memory unit consists of the data areas of the modules which make up the machine.

The interpreter is the same for all machines in the system since the language for all machines is the same high-level language with data type extensions represented by the modules on the particular machine. It can thus, with reference to the module abstraction, determine correctness of input and then, with reference to the module realization, cause the resource of a module to be executed. The interpreter virtually belongs to a machine; however, implementationally, there is only one interpreter and it is passed a machine pointer when its execution is required.

The machine has an address space associated with it in which module instances created upon this machine are allocated. Module instances are created upon this machine if a user creates an instance while attached to this machine or if a module instance is declared as separate by a machine running on this machine. Contained module instances created by a super machine are imbedded physically in the data area of the module which creates the contained instance (section 3.4.3). In addition to the module instances, the module definitions for modules of this machine are allocated within this address space.

The operations unit is represented by a single table which contains the names of the modules defined for this machine along with the addresses of their definitions. In addition, to maintain a record of instances allocated within this address space, a table of instances is also maintained.

Thus a machine is represented by an address space which contains module definitions and instances and tables to find these, and a virtual interpreter for the implementation language.

### 3.4.2 THE DEVELOPMENT ENVIRONMENT

#### 3.4.2.1 MACHINE DEVELOPMENT

There is a predefined machine in the system which permits machine creation and development. If permitted access to this machine and once attached to it, a user may declare new machines and develop modules for them. This same machine can be used to access machines to modify their repertoire or change abstractions or implementations of modules which are part of the machine.

Machines are maintained in heirarchical directories which may be created and added to with appropriate access rights. Typically, each user group would have a main directory entry under which it may develop systems. Thus a machine name is a multi part name which specifies the path in the directory and the directory name.

Development of a system may proceed as follows. The system designer first attaches to the machine development machine. He then creates a new machine for the system by invoking the operation provided for this purpose. In creating the machine he supplies a machine name (indicating the directory and where the machine is to be added in the directory) and may also supply access rights information for this machine (including run access, list access, environ access, and modification access). He may then develop the abstractions for this machine by performing operations which allow creation of modules and modification of module components namely the module definition, defines specification and uses specification.

In giving the module definition, the designer supplies the name of the environ for the module. When the module is first used (or when specifically instructed by the designer) the directory specified will be searched for this machine name during the consistency check for the module definition. At that time, the environ must exist and the module creator (i.e. designer) must have environ access (permission to use the machine as an environ) to the machine which is the indicated environ.

Once the modules for this machine have been designed, the designer may create any new machines required to implement the modules of the higher level machine, and then design the modules of those machines. This process may terminate at any time and any module which has not had an environ specified is assumed to have the hardware machine as its environ.

When implementation is to begin, the implementor attaches to the machine development machine. He may then list the modules to be implemented in the system (i.e. the machine which provides the system). To do this he must have list access to that machine. He may then select the module he is to implement and perform the implementation by modifying the module representation and implementation. Again, to do this, he must have modification access to this module.

When the module is to be implemented in an environ, the implementor may list the modules in the environ and the abstractions for any module he requires in his implementation (here he needs list access to the machine which is the environ and its modules). The implementation may then proceed. If the implementor wishes to develop a new environ for a module for which the designer has not specified one, he may do so if he has modify access to the module definition. He creates the new machine and the modules which comprise it just as the system designer did at a higher level.

#### 3.4.2.2 MACHINE USE

When a user connects to the DEMOS system, he is automaticly attached to a machine which provides two facilities:

the ability to list the machines available to the user and the ability to attach to a machine to which the user has run access.

When a user is attached to a machine, he may use any of the facilities provided by that machine, namely create objects of types defined on the machine and perform operations on those objects. These activities are carried out using the implementation language for the system just as the development of machines and modules is done.

When the user is finished working on one machine, he may end his session on that machine and return to the machine access machine. Here he may either end his session on DEMOS or attach to another machine to which he has run access.

### 3.4.3 MACHINES AND ADDRESS SPACES

Machines have a close connection with address spaces and the addressing mechanism (discussed in section 4.2). Machines provide address spaces in which module instances may be created and define the address allocation and resolution mechanisms to be used for any instance of a module of this machine which is an address space.

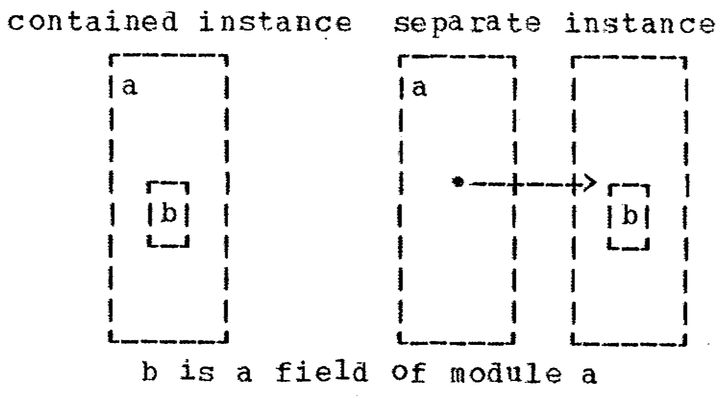


fig. 3.4.3.a Contained and Separate Instances

Module instances may be separated into two categories: separate and contained (figure 3.4.3.a). A contained module instance is one which is allocated as a part of the data area representing the instance of the module of which it is a field. A separate module instance is one which is created in a separate address space from the instance of which it is a field. When creating an instance, the creator declares whether the instance of the module is to be separate or contained. This is a property of a particular module instance and may be different for different instances of the same module.

Separate module instances are allocated within the address space of the machine of which the module definition is a part. All machines are created such that their address spaces are subspaces of the system space (section 4.2.3). This means that the address resolution activity is short in duration for separate instances. In addition, since memory management segments correspond to subspaces of the system space (Section 4.3.1), this forces these instances to be in different memory management segments with the advantage of making the segments smaller and the disadvantage of breaking the physical contiguity of the instances which might cause an extra segment load for the access of informtion in this instance.

A machine, when created, has one module predeclared. This is the address management module which is implemented upon a predeclared address management machine. If the creator of this machine desires, he may modify this module to provide a different address management facility. Any address space which is managed by a modified memory management module is called a used defined address space (section 4.2.5). When a module which is an address space (i.e. a dynamic data area) is manipulated upon this machine, it is this address management facility which is used to handle address allocation and mapping.

The module which implements the address management facility of a machine is a special type of module. Unlike all other modules, the address space module handles its own instance as a subspace of the address space it is managing.

It also has available to it low level routines for manipulating addresses and block tables.

The module specification and defines specification for the address space module are predefined and unchangable by a user. The other components may be modified by a user to perform a different address allocation/mapping function. A sample address space module is found in appendix A.

# 4 PHYSICAL ORGANIZATION OF DEMOS

## 4.1 DATA MANAGEMENT

### 4.1.1 DATA AREAS

Every entity in the DEMOS environment is an instance of a module and also part of another such an instance. This is true of all entities in the DEMOS system and all entities in programs developed under it. The instance of a module is the data upon which the implementation of the module operates. The code generated for the module implementation treats the data area as a storage area with initial address of zero. The addresses within the data area are converted into real storage addresses at execution time by the system (see sections 4.2.3 and 4.3.2).

To permit consistency checking, each data area consists of three fields:

1) instance pointer

2) creation date

3) data

The existence of the first two fields is known only to the elemental system routines which create, check and manipulate data areas. The compiler ensures that the code implementing the module does not reference these fields. The last field is the contents of the data area and, of course, is accessible to the code for the module implementation.

The instance pointer is a system address which points to the module which defines the abstract type of which this data area is an instance. This enables a check for consistency of the data area with its defining module (section 3.3.3.2).

The creation date is the date upon which this data area was created as an instance of its defining module. It is used by the system routines to determine if the data area has become inconsistent with its definition (section 3.3.3.2). It is initialized to the current date when the data area is created.

Modules (and hence their instances) can be separated into two categories. This separation is by the kind of data type they define. If the data type the module defines is dynamic in size over time, the module is termed dynamic. If the data type is fixed in size over time, the module is termed static. This is a property of the module (i.e. all instances) not just the property of a particular instance.

The size dynamism of a module can be determined by inspection of the module representation. A module is dynamic in size if any fields in the module representation are dynamic in size or are references to dynamically allocated objects. The first can be determined by inspection of the module definitions of the modules which define the types of the fields and the second can be determined from the field declarations directly (i.e. if the field is of type "pointer to").

An instance of a module which is static in size is represented by a simple data area. The area is fixed in size and all references can be made directly into it without having to go through any address resolution at this level.

An instance of a dynamic module is represented by a data area with two parts. The first part is a static area, just like that for a static module, in which all static data is maintained. The second is a dynamic area which contains all dynamic data. The first field in the static area is a block table for the subspace which is the instance of the address space module which manages this area. A data area of this type is called an address space. The two types of data areas are shown in figure 4.1.1.a

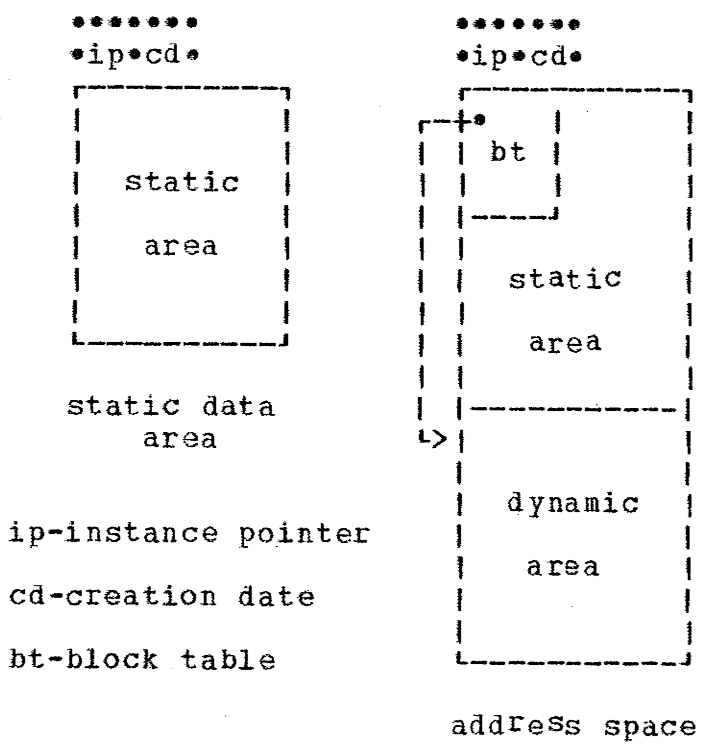


fig. 4.1.1.a Data Area Types

When reference is made to objects within the static portion, the offset within the data area is used. When reference is made to objects within the dynamic portion, the subspace identifier is used. When a transfer is made to a module in the base machine for this one, the same address resolution routines as those for the current data area are used if the module is static. However, if the module is dynamic, the address resolution routines for the machine of which the invoking module is a part, are used to resolve references into the instance.

Elemental entities, that is those defined as intrinsic in the system, are not represented by data areas themselves but are always part of some data area. Since they are not data areas, their use does not involve the overhead associated with data areas. This, however, prevents them from being consistency checked with their definitions since they have no instance pointer, but, because they are elemental, this presents no problems. All other module instances are represented by data areas as described above.

## 4.2 ADDRESS MANAGEMENT

### 4.2.1 ADDRESS SPACES

The method of successive decomposition for program development in addition to imposing a structure on the program, also imposes a structure upon the data. Modules at one level in the decomposition are defined in terms of the modules at the next lower level. The instance of a module defined at one level is thus composed of instances of modules defined at lower levels, and so on, yielding a heirarchical structure of the data in which the data areas of one level are imbedded within the data areas of another, higher level.

To the code for the module implementation, the data area representing the module instance is a set of storage locations labeled by a sequence of addresses. At the same time, many data areas are dynamic in size and, since a number of dynamic data areas may be imbedded within another, some method of address allocation must be provided to assign addresses within the outer data area to the dynamic portions of the imbeded data areas.

Two seemingly confilcting requirements thus arise. To allow for efficient implementation of dynamic data areas by retaining fixed addresses and avoiding copying, it is necessary for the dynamic portions to be non-contiguous within the outer data area. At the same time, however, to the code for the module implementation the data area must seem contiguous. The address space concept in DEMOS is the mechanism which resolves this confilct.

All data areas which are dynamic are created as address spaces (section 4.1.1). An address space provides for allocation of possibly non-contiguous addresses to dynamic portions of imbedded data areas as well as dynamic address translation so that, when these areas are referred to as contiguous by the module implementation, the correct reference will be made.

The address allocation and resolution performed for an address space involves some overhead. To enable this overhead to be avoided, a data area which is static is not created as an address space. When this is the case, no dynamic address allocation is done for imbedded data areas and no address translation is needed in order to reference the imbedded data areas.

Since data areas may be imbedded to an arbitrary depth, an address space may have imbedded within it further address spaces. All data areas (whether they are address spaces or not) which are imbedded within an address space are called subspaces.

The address space concept in DEMOS is similar to the concept of a segment as commonly used in virtual memory systems [2], [7] except that it extends the concept to allow dynamic address translation to be applied to subspaces (subsegments) as well as top level address spaces (segments). Thus a top level address space (system space) is equivalent to a segment in common virtual memory systems.

### 4.2.2 ADDRESS ALLOCATION

To enable address allocation, each address space is divided into a number of equal sized storage allocation blocks. When needed, the address space allocates addresses by blocks to its subspaces. The size of the storage allocation block for a subspace which is an address space should be an even divisor of the size of the storage allocation block of its superspace since, if it is not, inefficiencies would arise due to address space fragmentation.

Address allocation is only done to perform an extension to the length of a subspace. That is, the allocated addresses become addresses logically contiguous to the end of the subspace. The extension could be caused by an explicit request of a subspace or by the implicit request caused by a reference to a logical address beyond the end of the subspace. Each address space mechanism provides a routine to be used for extension of a subspace (section 4.2.5).

To enable the address space to keep track of the storage allocation and to enable dynamic address translation, the address space maintains a block table for each subspace. This table contains, in order by logical address of the block, the address space address of the storage allocated to the block. Thus, the address space address of the block containing the logical addresses in the subspace beginning at address zero is contained in the first entry of the block table for that subspace, etc.. Since the subspaces vary in length, the size of the block tables may also vary in length. To enable the address space to handle this, it maintains a subspace table which contains, for each sub-space, the length of the subspace and a pointer to the block table for the subspace.

It is at the address space level that DEMOS applies access protection. This is in addition to type checking since type checking verifies that the access has a valid form where access rights checking verifies that the user has been permitted access. Each subspace of an address space has associated with it an access list. This list indicates the type of access each user in the system is allowed to this subspace. The effects of denial of access are cumula-tive, that is, if a user is denied access to an address space he cannot gain access to any of the address space's subspaces.

To enable access rights checking, the address space maintains the access list associated with each subspace and a pointer to the list is maintained in the subspace table (see section 4.2.3).

### 4.2.3 ADDRESS RESOLUTION

To make possible dynamic allocation of addresses by an address space, the system performs address resolution at execution time. The information required for this resolution is maintained in the system space table (SST). The SST contains, for each active subspace in the system, a number of entries including:

1) the length of the subspace
2) the size of the address allocation block in units of which the subspace has been allocated
3) a pointer to the block table for the subspace

Address resolution proceeds as follows. When an address within a subspace is referenced, the entry in the SST for that subspace is interrogated. The address is divided by the allocation block length from the SST entry to give the logical block number while the remainder gives the offset within that block. The logical block number is used as an index into the block table pointed to by the SST entry. The block table entry then yields the superspace address of the beginning of this block. To this value, the offset within the logical block may be added to get the address within the superspace. If the superspace is not the system address space, the resolution continues, using the entry in the SST for the superspace (whose SST entry is found by a pointer from this SST entry, see section 4.2.4), and so on until the resolution yields an address within the system space (hereafter called a system address). These operations are similar to those involved in paging in a virtual memory system [7].

To enable address resolution to proceed to the superspace, the system maintains a pointer in the SST from the entry for each subspace to the entry for its superspace.

To avoid re-resolution of addresses which have not changed since last reference an extra bit is maintained in each block table entry. This bit (the resolved bit) indicates if this block has had its address resolved within the system space or not. Whenever a block table entry is interrogated, the resolved bit is first checked. If it is set, the address in the block table entry is the system address of the block and, if not, it is the address of the block within the superspace.

Whenever, during address resolution, a block is discovered with the resolved bit clear in its block table entry, resolution proceeds to the superspace for the subspace and so on, until it is finally resolved. Once resolved, the appropriate system addresses are placed into the block table entries encountered during the address resolution and the resolved bits are set. Whenever an address space modifies the address allocation for any of its subspaces, the resolved bit is cleared to force re-resolution of the modified address.

### 4.2.4 ADDRESSING SCHEME

During normal execution, references are usually made to one of three subspaces. These are the subspace containing the code for the module implementation which is currently being executed, the subspace containing the data area which is the module instance and the subspace containing the execution stack. To enable these subspaces to be easily referenced, the system maintains three pointers into the SST. These are the instruction space register (ISR), the data space register (DSR) and the stack space register

(SSR). They each contain a pointer to the SST entry for each of the three current subspaces.

Since the data area actually being referenced may be either a subspace (if it is imbedded within an address space) or part of a subspace (if it is imbedded within a data area which is not an address space), an additional value is needed to locate the actual data area. This is the offset of the actual data area within the subspace indicated by the space register. This is maintained in a register, one for each current data area. There are three offset registers, the instruction offset register (IOR), the data offset register (DOR) and the stack offset register (SOR). The offset registers and space registers are maintained in pairs, the ISR with the IOR, the DSR with the DOR and the SSR with the SOR.

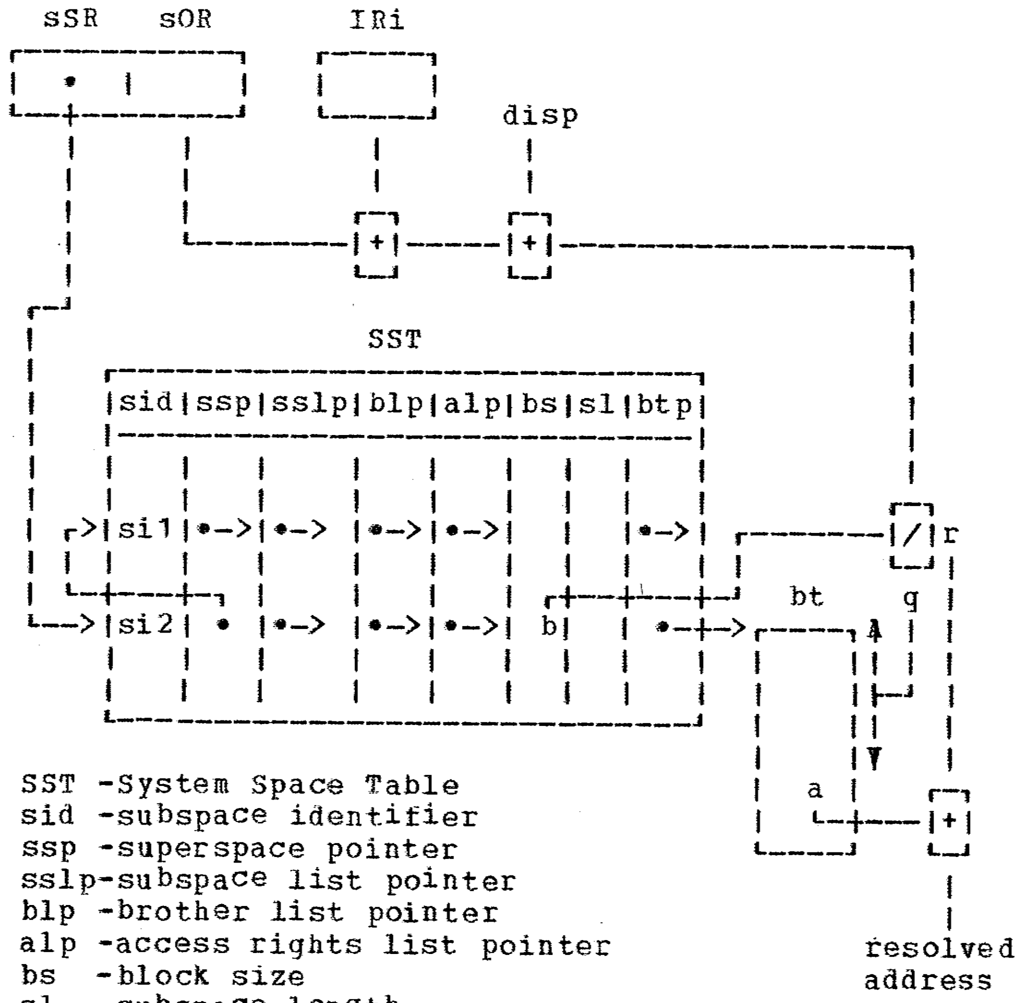
An address reference consists, then, of three parts. The first is a data area reference (indicating the instruction, data or stack space), the second is the displacement within the indicated data area and the third indicates an index register. The subspace address is then computed as:

$$(sOR)+(IRi)+disp$$

where "s" is I for the instruction area, D for the data area, and S for the stack area; IRi is the i'th index register and "disp" is the displacement. This address then undergoes address resolution as explained in section 4.2.3 The complete address resolution activity is shown in figure 4.2.4.a

An invocation of an resource provided by another module does not fit into the above addressing scheme. This type of reference involves a change in all three of the data areas. The instruction data area is that for a different module implementation, the data area is that for a different module instance, and the stack data area is a new stack frame.

A special system call is used to handle inter module invocations. To it are passed the system names of the subspaces containing the module implementation, the subspace containing the module instance and the offsets of the data areas within those subspaces. These names are used to search the SST for the entry for the subspaces. The system routine searches the SST for the names, verifies that the access is valid and that the date stamps are consistent and then allocates a new stack frame. It then stores the current (ISR,IOR), (DSR,DOR), position of the old stack frame as well as the other registers, etc., in the new stack frame, which has been allocated as a subspace of the current stack address space. Next it copies the parameters into a



SST -System Space Table
sid -subspace identifier
ssp -superspace pointer
sslp-subspace list pointer
blp -brother list pointer
alp -access rights list pointer
bs -block size
sl -subspace length
btp -block table pointer

| | |
|---|---|
| si1 -a subspace identifier | b -a block size |
| si2 -a subspace identifier | bt-a block table |
| sSR -space register s | a -a superspace address |
| sOR -offset register s | q -quotient |
| IRi -index register i | r -remainder |
| disp-displacement | |

fig. 4.2.4.a Address Resolution

parameter area in the new stack frame. The ISR and DSR are set to point to the SST entries for the subspaces which were given as arguments to the system routine. The IOR and DOR are set to the offsets also supplied. The system then gives control to the module implementation which was to be invoked.

For return, again a special system call is made. The system then recovers the (ISR,IOR), (DSR,DOR), (SSR,SOR) and registers from the stack and then deallocates the stack frame.

The use map contains addressing information for use in referencing external resources (section 3.2.1.2). It contains the offset within the subspace of the resource being referenced, while the object code contains the system name of the subspace.

Since the use map is referenced frequently, the address space containing the data area which is the use map for the currently executing module implementation is noted by a fourth space register-offset register pair. These registers are called the external space register (ESR) and the external offset register (EOR). The ESR points to the SST entry for the subspace containing the use map and the EOR contains the offset of the use map within that subspace. This pair is maintained by the system in the same way that the (ISR,IOR), (DSR,DOR) and (SSR,SOR) pairs are. The system call for a branch to another module instance is passed then three system name-offset pairs for the new (ISR,IOR), (DSR,DOR) and (ESR,EOR).

A system name is a path in the system space tree from either the current node or the root to a data area. Each branch selector is a subspace identifier. When a subspace is created within an address space, a routine in the address space is invoked to create a unique subspace identifier within that address space. The subspace identifier is thereafter the selector for the branch from the address space to the subspace. When a subspace is destroyed, a routine in the subspace's superspace (an address space) is invoked to return the subspace identifier to the address space for use in creating future subspaces.

The system name for the instruction space always describes a path from the root node. The system name for the data space usually describes a path from the current data space node. It can in certain circumstances describe a path from the root node (i.e. in the case of a separate module instance, see section 3.4.3).

To enable the search for subspaces by path name, the SST is organized as a tree. Each entry has two additional fields, one a pointer to the entry for one of the space's subspaces, and the other a pointer to the SST entry for one of its brothers as subspaces of their common superspace. In addition there is a field which contains the subspace identifier of the space as a subspace of its superspace. Depending on the search required, the search begins at the root (or current) node and follows the pointers checking the subspace identifiers until it reaches the desired node.

To save space in the system space table, the positions for entries are multiplexed among all possible entries. During a search for a node in the tree, if a null pointer is encountered, the system adds an entry to the SST by invoking a routine in the superspace of the node it is currently at (if it is currently searching a brother list) or in the current subspace (if it is attempting to descend to a subspace) which returns (upon being passed the subspace identifier) the access list pointer, the block table pointer, the subspace length and the subspace address allocation block size. These values are placed into the SST and the values for subspace identifier and superspace pointer are set, the subspace pointer and brother pointer are set to null and the brother pointer (if currently searching a brother list) or the subspace pointer (if attempting to descend to a subspace) of the brother or superspace currently at, is updated.

To free up SST entries, leaves may be pruned from the tree. When a leaf is pruned, it is possible that a stacked ISR, DSR or ESR points to the SST entry. A stacked SSR will never point to a leaf to be pruned since stack frames are always allocated as subspaces of the current stack frame which ensures that all stacked SSRs point to superspaces of the current stack space and are thus not pointing to leaves in the space tree. To ensure that the stacked values do not point to invalid SST entries after a pruning, the SST entry contains a pointer to the first entry in a connection list. A connection list is a list of the stack entries for space registers that point to an SST entry, linked by pointers. When an space register is stacked, it is stacked along with the current value of the connection list pointer, and the connection list pointer is pointed to the stack entry.

When a leaf is pruned, the connection list pointer is followed to find all stack entries containing pointers to this SST entry. These stack entries are replaced by the system name for the subspace. When a space register is unstacked, if the entry is a system name, the name is resolved within the SST and the new SST pointer is placed into the space register.

## 4.2.5 USER DEFINED ADDRESS SPACES

To allow flexibility in the manner in which addresses are allocated, users are allowed to define an address space mechanism. This mechanism must provide the same resources as the standard address space mechanism, but may use any method of address allocation desired by the user.

The user defined address space is implemented as a module of a machine (section 3.4.3) and must provide four routines in addition to the module creation and disposal routines that all modules must supply. An example of such a module is given in appendix A. The additional routines are:

1) CREATE
2) DESTROY
3) EXTEND
4) FETCH

which are used to create a new subspace under the user defined address space, remove a subspace so created, extend the length of such a subspace and return information necessary to the system for subspace management.

The CREATE routine is invoked whenever a subspace is to be created under the user defined address space. The access rights list is passed to the routine so it may initialize the access rights for the subspace. The user defined address space must build a block table for the new subspace and return a unique (within the address space) subspace identifier for that subspace.

The DESTROY routine is invoked whenever a subspace within the user defined address space is to be removed from the system. The subspace identifier is passed to the DESTROY routine to identify the subspace to be removed. The user defined address space may then remove the block table for the subspace, recover the addresses allocated to the subspace, and note that the subspace identifier is now reusable.

The EXTEND routine is invoked whenever a subspace of the user defined address space is to be extended beyond the space already allocated to it. The invocation occurs only if the user causing the extend request has extend access rights to the subspace. The subspace identifier is passed to the EXTEND routine to identify the subspace to be extended. The user defined address space must allocate a block of addresses to the the subspace and note the allocation in the block table. If the allocation cannot be done, the user defined address space must return an indication to that effect.

The FETCH routine is invoked by the system inter module transfer routine when, upon attempting to reference a subspace, it is detected that there is no entry for that subspace in the SST. The system then passes the subspace identifier to the user defined address space which is the superspace of the desired subspace. The FETCH routine must return the required information to the system. This information includes the pointer to the subspace's access rights list, the size of the block in which addresses are allocated to the subspace, the length (in blocks) of the subspace, and a pointer to the block table for the subspace. With this information the system can create the SST entry for the subspace and enable address resolution to proceed.

When an address space is to be referenced either to create the address space in the first place or to access a subspace of it, the system preloads the SST entry for subspace zero of this address space with temporary values. It sets the subspace identifier to zero, the superspace pointer to point to the SST entry for the address space itself (this entry is already in the SST), the brother pointer to nil, the access rights list pointer to an access rights list granting unrestricted access, the block size to the maximum value possible, the subspace length to one (block), and the block table pointer to point to the beginning of the data area (i.e. to the first field in the data area, which is the block table for subspace zero).

If the address space itself is being created, the system loads the first entry of the block table for subspace zero with the address of the dynamic area (called the Base of Dynamic Area - BDA). It then invokes the address space module's creation routine. This routine must fill in the appropriate entries in its subspace table for subspace zero (which is the address space module instance itself) and take any other initialization action it desires.

Once the creation routine completes, or if this is a reference to a subspace, the system then does a fetch on subspace zero to obtain the current values for the access list, block length and subspace length. The block table pointer returned by FETCH is ignored since the block table for subspace zero must be the one at the beginning of the data area. The system is now ready to perform the access to any subspace of the address space by using CREATE, DESTROY, EXTEND, or FETCH.

The creation routine and the module representation for an address space module must each adhere to one restriction. The representation must cause allocation of no more than one block in the address space at creation and the creation routine may only access variables allocated within this block. This is due to the fact that the system preload has provided for only one block in subspace zero (since it cannot determine where another block would reside), and until the fetch for subspace zero is done, no other blocks can be referenced. In addition, the fetch for subspace zero must only reference within block one of subspace zero for the same reason.

One final restriction is imposed on the CREATE routine. Whenever it creates a new block table, it must allocate it as a separate instance (section 3.4.3). If this is not done, the block table would be allocated as a subspace of this address space and this would cause a resursive invocation of the CREATE routine to create the subspace for the block table. An added advantage of having the block tables as separate instances is that then there would be a memory management segment which contains all of the block tables for address spaces created by this address space mechanism. This segment is likely to remain resident, removing the posibility of extra memory management operations upon reference to a block table.

To ensure that the appropriate address management routines are used for an address space, all address management operations are initiated via the system. When an inter module invocation is made, the inter module transfer routine determines if the data area involved in the transfer is a separate or contained instance. If it is a contained one, the routine loads the addresses of the address management routines for the current machine for use by the system create, destroy, extend and fetch routines. Otherwise the appropriate addresses for the address mechanism of the separate address space are loaded. Whenever the user defined create, destroy, extend or fetch routines are invoked by a system routine, they are invoked with subspace zero of the address space as their module instance. In this subspace, all the data required by these routines to perform their task must be located. All user initiation of create, destroy and extend is done via a call to a system routine which in turn invokes the appropriate user defined address management routine with the appropriate parameters.

## 4.3 MEMORY MANAGEMENT

### 4.3.1 MEMORY SPACES

Memory management in DEMOS is based on a paged segmented system as are MULTICS [2] and other systems. This provides a virtual memory system with efficient use of space and minimal delay when addresses referenced are not resident in main memory.

As in MULTICS, all data potentially referencable by programs in the system have unique addresses within the system called system addresses. In DEMOS these addresses are a direct result of the compilation and address resolution processes, and are provided in a single form to the memory management system. The system need only concern itself with the address decoding and the location of the actual data within the storage hierarchy, instead of being concerned with data management.

The entire collection of storage devices in the system is ordered into a hierarchy by access time and storage capacity with the devices with short access time and small capacity at the top of the hierarchy, and those with longer access times and larger capacity at the bottom of the hierarchy. Each level of the hierarchy is called a memory space.

Each memory space is managed in a similar manner. When a request for a piece of data is made to the memory space, it determines if it currently contains the data requested and, if so, returns that data. If the requested data is not currently within the memory space, it makes a request for the data to the memory space immediately beneath it in the hierarchy. The top level memory space gets its requests for data directly from the CPU and the lowest level memory space contains all data accessable to the system and thus never has to request data from a lower level.

Memory management is handled separately from address management (section 4.2) and data management (section 4.1), however the memory management segments are equivalent to the subspaces of the system space in address management. This forces physical (i.e. within the memory management system) contiguity of areas which are logically contiguous, which enables the memory management system to take advantage of the locality of reference which manifests itself in the references to the logical address spaces. This also allows the system subspace identifier to be used as the segment identifier in the memory management system yielding a simple correspondence between system subspaces and memory management segments.

Each memory space is divided into a number of equal sized page frames. The page frame size of a memory space at one level is at least as large as that of the next higher level and is likely a multiple of that size. The actual page frame sizes are chosen according to access times and transfer rates of the storage devices on which the memory space is located [7].

Each segment is physically divided into pages at each memory space level. These pages are of fixed size (within the level) and equal to the size of the page frames at that level. The segments are the same throughout the hierarchy.

Whenever pages of a segment are in the memory space at one level, there is at least one page of that segment in the memory space at the next lower level. This requires that the number of segments which may have pages in the memory spaces must not decrease as one proceeds down the memory space hierarchy. In addition, for each page in the memory space at one level, there must be room for the page (of the next lower level) which contains the same data as the page at the higher level. This requires that the number of page frames must not decrease as one proceeds down the memory space hierarchy. These two provisions provide a linear storage hierarchy and make memory space page and segment mulitplexing, which is performed at all levels, more efficient (section 4.3.2).

### 4.3.2 ADDRESS DECODING

Since pages of segments are placed arbitrarily into page frames in a memory space, and the number of page frames in each memory space (except the one at the lowest level) is limited and less than the number needed to hold all of the pages of all of the segments in the system, it is necessary to multiplex the page frames and to provide a scheme for mapping system addresses into memory space addresses and providing that the correct page of the segment is actually in the memory space page frame.

Page frames in each of the memory spaces (except the one at the lowest level) are multiplexed over pages in all segments accessable to the system. In each memory space, only some of the pages are maintained in page frames. Whenever it is necessary to access a page which is not currently allocated to a page frame, a page is selected according to a paging strategy (eg. LRU), is removed from the memory space and its page frame is allocated to the page in question.

This page may then be moved into the page frame from the next lower memory space.

To maintain a record of all segments currently having pages in the memory space, the memory spaces each maintain a contained segment table (CST) which contains, for each segment with pages currently in the memory space:

1) the segment identifier
2) a page table
3) fields for use in determining segment in/out criteria

The segment identifier is the intra memory space identifier for the segment and is the same as the subspace identifier for the system subspace to which it is equivalent.

To allow a mapping from pages within a segment to page frames within the memory space, the memory space maintains, for each segment in the memory space, a page table. All page tables in a memory space are of the same length and are large enough to provide mapping for a segment of maximum size. Since there are, in general, more pages in all segments which are contained in a memory space than there are page frames in a memory space, some of the page table entries must be for pages which have not been allocated page frames in the memory space. A flag is maintained in each page table entry to indicate if this page is currently allocated a page frame in the memory space.

Since page frames in the memory space are multiplexed over the pages in all segments, a scheme must be provided to indicate which pages are to be maintained in a certain memory space. Additional fields are maintained in the page table entries to record information to be used in determining which page should be paged out of the memory space when a page not currently allocated a page frame in the memory space is referenced. For example, a least recently used (LRU) page out algorithm might be used.

Since, at all times, when a page of a segment resides in a memory space at one level the page containing that page in the memory space at the next lower level also resides in that memory space (section 4.3.1), when a page is to be paged out of a memory space a copy operation to the next lower level is only required if the contents of the page has changed since it was fetched from the next lower memory space. A flag is maintained in the page table entry for the page to indicate when the copy needs to be done. The flag is set whenever a copy (store) operation is performed into this page at this level. On page out, if the flag is set, the copy is performed to the next lower memory space (causing the flag on the page table entry there to be set). If the flag is not set, the copy need not be performed since the page has not changed since it was fetched. In addition, when the copy operation must be done, the segment is guaranteed to be contained in the next lower memory space (section 4.3.1) so no delay due to a segment fault can occur.

Address decoding begins when a fetch or store request is made to the memory space for a particular memory space address. This address is in the form of a segment identifier (number) and an offset. The offset is divided by the page size for the memory space to give the the page number and page offset. The CST is searched for the entry which contains the segment indicated by the segment number. If there is no such entry, a missing segment must be processed. When the entry is found, the page table for that entry is interrogated. If the page table entry for the page indicated by the page number indicates that the page is not in the memory space, a missing page must be processed. Otherwise the memory space location is found by adding the page offset to the address field of the page table entry and the fetch or store is performed from or to that location. This activity is shown in figure 4.3.2.a.

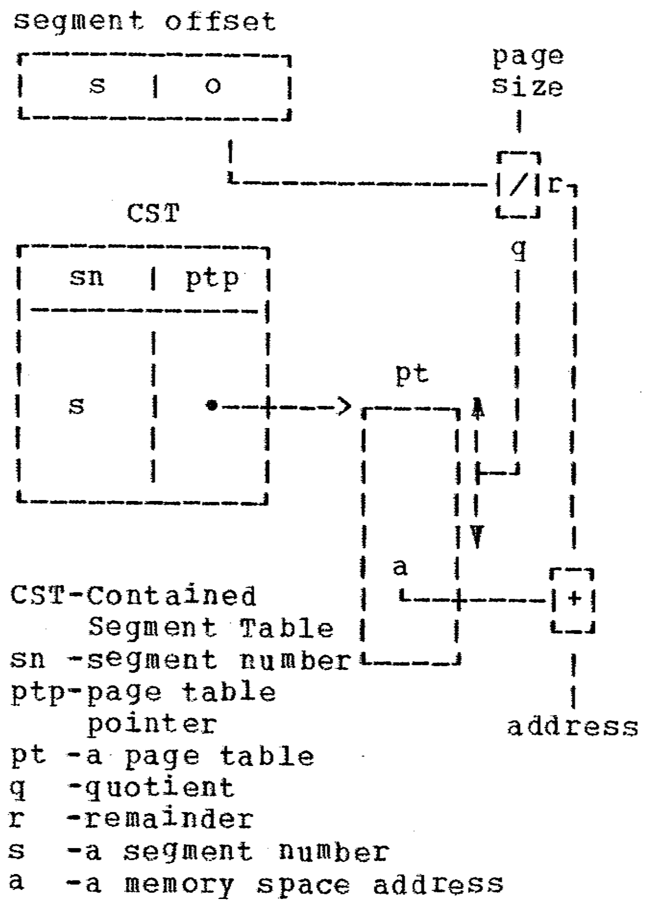


fig. 4.3.2.a Address Decoding

When a missing segment is discovered, the CST is searched for an entry which best meets the segment out criteria (i.e. least recently used and contains no pages). This entry is then used for the missing segment. The new segment number is placed into the entry, and, since the segment which was removed from the table had no pages in the memory space, the page table entries do not have to be marked as missing since they are already marked so.

When a missing page is discovered, the page tables in the CST are searched for the entry which best satisfies the page out criteria (i.e. least recently used). This entry is then marked as missing. The entry is then checked to see if the page has been modified since it was fetched. If so, a store operation is initiated in the lower level memory space. Since this store operation may take some time, the task which initiated this address decode is suspended pending the termination of the store. When the transfer is complete (of if it wasn't necessary), the page frame allocated to the page table entry is freed.

Now there is a free page frame and the segment page encountering the missing page condition may be allocated a page frame for the page. The frame address is placed into the table entry. The page is then loaded into the page frame by requesting a fetch of the required page from the next lower memory space. Since again this takes some time, the task initiating this address decode is suspended pending the termination of the fetch. Since some other process running in the system may request the same page before this fetch is complete, steps must be taken to prevent an unnecessary page fetch.

Before the fetch is initiated, another bit (the loading bit) in the page table entry is set. Whenever a missing page is processed, this bit is checked. If it is set, the task causing the address decode is suspended pending the termination of the fetch of the page. When the fetch of the page is complete, the missing page process for the task causing the fetch resets the loading bit, marks the page as being in the memory space, and notifies any other tasks waiting for the fetch to terminate that the fetch has indeed terminated. The missing page processes for the other suspended tasks which are awaiting such notification, then terminate as if they had caused the page to be fetched.

The amount of data transferred in a fetch or store is always the number of bytes in the page frame size of the higher of the two memory spaces involved in the transfer. This is due to the fact that, from the point of view of the higher of the two memory spaces, the operation is either a page in or page out. Since this is true, the offset needed for the fetch or store from or to the lower memory space is just the page number of the page in the higher memory space, and thus the page frame size of the lower memory space is recorded as the number of higher level memory space pages that can be stored in one lower level memory space page frame.

At the top of the memory hierarchy is the memory space which is accessed directly by the CPU. This space is handled as if fetch and store requests come from a higher level memory space whose page size is the width of the data path between this memory and the CPU.

## 4.4 HARDWARE SUPPORT

To improve the efficiency of the address resolution and address decode operations, certain functions are relegated to hardware.

The space registers contain two pieces of information:

1) the access rights
2) a pointer to the SST entry

The access rights field contains the access rights of the user whose process owns the space register, to the subspace at which the space register points. The access rights stored in the space register are the access rights which were in effect when the task the user started first attempts to reference the subspace. These rights will remain in effect until the task terminates even if the user's access rights are modified in the subspace's access rights list during the life of the task since problems would arise if a user is denied access to a subspace after partial completion of some activity in that subspace.

The pointer to the space table is just an offset within the SST with which the space table information may be accessed. It is guaranteed to point to a valid SST entry since the SST entry for a subspace contains a field indicating the number of space registers of tasks being currently executed which point to the SST entry. Unless this count is zero, the SST entry cannot be pruned from the table (see section 4.2.4).

To improve the speed of address resolution, the hardware performs part of the resolution operation. It first verifies that the access is valid by comparing the access requested with the access rights in the space register. If the access is invalid, it faults to a system access rights fault routine. The hardware then computes the offset within the subspace by adding the offset register, index register and displacement from the instruction. From this it computes the block number and block offset within the subspace by referencing the SST entry pointed to by the space register and accessing the block length field.

If the block number exceeds the subspace length indicated in the SST entry, the hardware either invokes the system length fault routine (if the user does not have extend access to the subspace) or to the system extend fault routine (if the user does have extend access to the subspace). Next, the hardware interrogates the block table entry for the block number developed earlier. If the resolved bit is not set, it faults to the system resolution fault routine. If the resolved bit is set, it develops the system address by adding the block offset to the address found in the block table entry.

Part of the system address decode is also handled by hardware. The CST for the top level memory space is maintained in associative memory. This enables an efficient search of the CST by the hardware.

The hardware takes the system address developed in the address resolution phase and extracts the segment number and segment offset. It then divides the segment offset by the top level memory space page frame size to yield the page number and page offset. It then searches the CST for the top level memory space (in associative memory) for the entry with the given segment number. If there is no such entry, it faults to the system segment fault routine.

The hardware then interrogates the entry for the page number in the page table of the CST entry. If the missing page flag is set, it faults to the system missing page fault routine. If the flag is clear, it computes the machine address as the sum of the page offset and the address found in the page table entry.

To allow the hardware to process some of the address resolution and address decode functions, the system must provide six fault routines. These are:

1) access rights fault routine
2) extend fault routine
3) length fault routine
4) resolution fault routine
5) missing segment fault routine
6) missing page fault routine

The access rights fault and length fault routines abort the task and notify the user why the task was aborted.

The extend fault routine determines which subspace the fault occurred for and then branches to the EXTEND routine for the address space which is the superspace for the faulting subspace. This enables a user defined address space to get control to allocate addresses to its subspaces. When the EXTEND routine returns, it indicates if the subspace was extended. If it was, the extent fault routine terminates. If not, it aborts the task indicating to the user why the task was aborted.

The resolution fault routine performs the address resoultion through the system space tree until it finally gets a system address for the referenced address. It can then terminate causing the access to be retried by the hardware.

The missing segment fault routine finds a CST entry which can be used for the segment information for the referenced segment. It then causes the appropriate information to be loaded into the CST entry and terminates allowing the reference to be retried by the hardware.

The missing page fault routine locates a free page frame and causes the referenced page to be loaded into the page frame. It updates the missing page bit and then terminates allowing the reference to be retried by the hardware.

The above hardware and software additions should make the address resolution and address decode operations reasonably efficient to enable rapid execution of code within the system.

# 5 DEMOS IN REVIEW

## 5.1 SATISFACTION OF THE GOALS

The basic goal behind the design of DEMOS was the desire to provide a software environment suitable for developing systems using a modular development methodology. An important consideration was that large projects are developed by a group of people working in parallel. This leads to the desire for independent development of parts of the system. In addition, it was recognized that design should precede implementation in a project and that, in a multi-level system, design of at least the first few levels should proceed before any implementation begins.

Even though development of the system proceeds with independent efforts being made by several developers at the same time, the parts of the system each is developing are not totally independent. These parts interact via interfaces between them to make the system a whole. In independent development, the enforcement of proper use of these interfaces is probably the biggest problem facing project managers.

When designing a system, a major consideration is the design of the interfaces between the parts. For the system to perform properly, it is imperative that this part of the design effort be done well. In addition, it is desirable for the specification of these interfaces to remain stable throughout the development of the project. Having the design debugged before the implementation begins goes a long way towards making this possible.

The basic goals for the DEMOS system were then: that design of a system could precede implementation and that the design could be verified before implementation begins; and that both the design and implementation phases could proceed as a number of parallel, independent activities but, at the same time, that the interactions between the parts under independent development could be maintained consistent with other parts and the design in general.

The development unit chosen for development under DEMOS was the abstract data type, which, in DEMOS, is represented by a module (section 3.1). This provided a logical unit for both design and implementation. To collect these units into a group suitable for the representation of a subsystem, the concept of an abstract machine (section 3.4) was introduced. This allowed the design to specify the tools available to an implementation by specifying a machine upon which a module was to be implemented (based).

So that the design could be verified automatically and that the implementation could be verified consistent with the design, the design was to be provided to DEMOS in the form of an abstraction (section 3.2.1.1) for a module. To enable the design of a module and its implementation to be carried out at different times, DEMOS retained the abstraction so that, when the implementation was done, the verification of its consistency with the design could be carried out automatically. In addition, this enabled the design of interacting modules to be done at different times and still allow automatic checkirg of the designs. Also it allows changes to be made to both the design and implementation and DEMOS can recheck the modified versions after the changes are made.

It was recognized that, in any system under development, modifications would be rife. This would imply that a module design and/or implementation, once verified to be correct within the system, could become incorrect through either a modification to it or to some module with which it interacted. This lead to the maintenance, within DEMOS, of dependencies between modules in the system (section 3.2.2) and the use of automatic recompilation, in the event of a modification, to perform checks of consistency between modules (section 3.3.2).

Since recompilation of modules is an expensive proposition, all efforts were made to reduce the number and extent of these recompilations. This brought about the division of a module into a number of components (section 3.2.1), so that modifications of one component of a module need not force recompilation of other components in this or other modules if this is not warranted to verify consistency. It was recognized that abstract machines, once developed, were a resource which could potentially be shared throughout the user community (much as subroutine libraries are now). With this in mind, it was realized that a user who is willing to provide a machine he has developed to other users should be able to specify which users may use the machine and how they could use it. This leads to the use of access rights for machines. Also, when the owner of a machine makes a change to the machine, he should not encounter added delays due to the fact that other users have developed modules based on his machine. This would occur if automatic recompilation of dependent modules was done immediately when a change is made to a providing module. To spread these delays out over the user community, DEMOS recognizes some dependencies between components to be different from others and the concept of deferred consistency checking evolved (section 3.3.3.1).

These developments provide the framework of a system which satisfies the goals set for DEMOS. However, it was still necessary to develop the underlying representation which would enable all these developments to be implemented.

When a module is actually used in a system, there must be some concrete representation of the module instance. The concept of a data area (section 4.1.1) provided this representation. It was recognized that the successive decomposition aspect of the development methodology lead to the logical imbedding of module instances within others. This logical imbedding was carried over into the physical representation of module instances.

At the same time, it was realized that module instances would not necessarily remain static in size since an abstract type may be such that the amount of information it represents may be dynamic over time (eg. an abstract type defining the concept of a file). Since module instances were physically imbedded within each other, this lead to a problem which was resolved by the concept of address spaces (section 4.2.1) as a means of allowing dynamic allocation of space within a module instance.

Since it may be desirable to allocate space in a module instance in different manners for different abstract types, DEMOS allows a user to specify a user defined address space mechanism (section 4.2.5). This allows a user to use any space allocation and mapping principle he desires within a specific module instance.

Since the imbedding of module instances within each other causes a segmenting of data into units which have the property that most references by a program to these units would be clustered in time, it was reasonable to use this knowledge to enhance the performance of accesses to data. This lead to a memory management scheme (section 4.3) using segmentation and paging, where segments were equated to certain address spaces (section 4.3.1) to allow the logical grouping of the data to reduce the load on the memory management system.

A last point covered was a discussion of the hardware requirements needed to support the address and memory management schemes devised at a reasonable level of efficiency (section 4.4).

When all these developments are brought together, DEMOS would provide a suitable environment for modular development of systems as outlined by the goals set for it and do so in a manner which should be reasonably efficient and convenient to use.

## 5.2 ADDITIONAL BENEFITS AND FUTURE DEVELOPMENT

The organization and representation of machines and modules in DEMOS which were designed to satisfy the specific goals of section 2.5 would also enable, at little extra cost, other features not in the original plan.

The first is a high-level interactive debugging package. This would be easy to incorporate since all the information required for interrogating the contents of data areas already exists, and there is already in the system a facility for interpretive execution. Since each data area is connected to its module definition (via its instance pointer), it is a simple task to discover the values of any fields in the data area. The symbol tables maintained in the module definition give the information required to find a field in a data area and, in addition, to display it in a meaningful form. This means that the debugging inquiries can be made at high level, that is using field names, and that responses (displays) can also be at a high level (i.e. appropriate to the type of the data item) instead of at a low level (e.g. hexadecimal).

Since there is already an interpreter available on all machines in the system, the debugging aid need only invoke this interpreter to interpretively execute operations from a module. The interpreter could also be used for the generation of high level display of data areas by invoking the output routines provided by a module for its data type.

A second feature which could be developed in DEMOS is a simulator for modules which have been defined, but not yet implemented, for testing purposes. The information retained in the defines specification could be augmented by a formal definition of the behaviour of the resources provided by the module. An additional resource could be provided by the "module" module which would simulate the behaviour of the module resources based on the formal definition of their behaviour. A scheme similar to this has been implemented in TOPD [13].

As techinques for automated program proving are improved, it would be possible to incorporate an automatic program prover into the system using a scheme very similar to that discussed above for the module simulator. Again a formal definition of the behaviour of module resources could be maintained as part of the defines specification. A program prover could use these specifications and those of the modules in the base machine for this module (pointed to by the module definition) to perform an automated proof of the functioning of this module and ultimately of the entire system under development.

One extension to the system which requires a major effort, is the addition of more programming languages to DEMOS. DEMOS could function sufficiently well with different languages being used for development except for the verification of correctness of interfaces between modules developed in different languages. The problem arises from trying to determine if the types specified at one end of the interface match the types used at the other end. This is a generalized mode equivalence problem (i.e. between, rather than within languages). If a standard representation for types could be developed and the translation for types in all languages into that standard representation was known, then modules in various languages could interact.

# APPENDIX A  ADDRESS SPACE MODULE EXAMPLE

The following is an example of an address space management module (see sections 3.4.3 and 4.2.5). The language used in this example is not any formally defined language but is instead an informal cross between Pascal and ALGOL68 with extensions to allow module definition, and is used only as a vehicle for the presentation of this example. It assumes the existence of a dynamic array type which is an array that has an initial allocation of a specified number of elements but can be extended by invocation of a procedure "extendarray" to add a number of elements to the end of the array. In addition, the standard procedure "sep" is like "new" in Pascal or **heap** in ALGOL68, but differs from new in that it creates the object as a separate instance instead of a contained one.

A module always provides two procedures called "new" and "dispose" which are invoked upon creation of an instance of the module and just prior to the destruction of an instance. The "new" routine has as parameters the parameters of the module (and is the only routine to which these parameters are directly available).

The machine "system.addressmachine" is a predeclared machine which provides types accesslist, address and blocktable. The procedure "load" accepts a variable and an address and loads the contents of the specified address into the variable. The procedure "store" accepts a value and an address and stores the value at the address specified. the operator ¬= accepts two address values and returns the boolean value TRUE if they are different addresses. The operator + accepts an address (a) and an integer (i) and returns an address value which is the address i storage units from the address a. The function "getaddr" accepts a block table (b) and an integer (i) and returns the address contained in the i'th entry of block table b. The procedure "setaddr" accepts a block table (b), an integer (i) and an address (a) and sets the i'th entry of block table b to contain the address a.

The module defines four routines called "create", "destroy", "extend" and "fetch". The requirements for these and described in section 4.2.5. It uses the resources: load, store, ¬= and + defined by module "address", and getaddr and setaddr defined by module "blocktable". In addition, it requires the existence of the access list type as defined by the module "accesslist".

The module is represented by (i.e. has as its instance) a pair of addresses and a dynamic array. The address "endspace" gives the address of the first location after the last logical block so far allocated (i.e. the address of the first location past the current end of the area). The address "nextblock" gives the address of the next block to be allocated. Free blocks are maintained as a linked list with the address of the next block in the list stored in the first location of a block. When nextblock equals endspace, there are no more free blocks. In this case the new block to be allocated will be at this address and reference to it may cause extension of this address space.

The dynamic array "spacetable" represents the space table for the address space. It is initially allocated 101 entries which will allow the initial representation of the address space module instance to be less than one block in length. The subspace identifier is used as an index to access the subspace information for that subspace in the space table. The subspace information for subspace zero (the instance of the address space module) is stored in entry zero of the array, guaranteeing that it is in block one of subspace zero as required.

The size of a storage allocation block (blklnth) is fixed as is the maximum number of blocks per subspace (btsize).

The address space instance creation routine (new) fills in the entry for the instance as subspace zero and then initializes the rest of the subspace table to free entries. An entry is free (i.e. no subspace exists with that index as subspace identifier) if there is no block table for that entry (i.e. the block table pointer is nil). It then sets endspace and nextblock to indicate that there are no free blocks available and that the first location for a new block immediately follows the first block of subspace zero (which resides at address bda).

The address space instance destruction routine (dispose) performs no action.

The subspace creation routine (create) searches the subspace table for a free entry. If none are found, it attempts to extend the subspace table. If that fails, a create is not possible and zero is returned as the subspace identifier indicating that create failed. If extension was possible, the new entries are initialized to free and the first one is the position of the entry for the new subspace.

If an entry was found, the information is entered into this entry and a block table (of size btsize) is created. The subspace identifier is the index of this entry.

The subspace destruction routine (destroy) adds all blocks allocated to the subspace to the free block list by storing the address nextblock into the first location of the block and then loading the blocks address into nextblock. It then frees the block table allocated to the subspace and marks the entry in the subspace table as free.

The subspace extension routine (entend) determines if the subspace has already reached maximum size (i.e. btsize blocks). If not, it allocates the next available block to the subspace as the next logical block and then, if the free block list was not empty, sets nextblock to indicate the next block in the list. Otherwise it sets both endspace and nextblock to the next location for a block to be allocated, that is the location after the end of the block just allocated.

The subspace information fetch routine (fetch) extracts the appropriate information from the subspace table and returns it. A fetch for subspace identifier zero will only access the first block in subspace zero satisfying the restriction on fetch.

The "code" for the module follows:

```
addressspace:MODULE(ssbtable:REF blocktable,
                alist:REFaccesslist, bda:address)
              ENVIRON system.addressmachine;

  ABSTRACTION

    DEFINES
      create:PROC(alist:REF accesslist):INT;
      destroy:PROC(id:INT);
      extend:PROC(id:INT):BOOL;
      fetch:PROC(id:INT):
                  STRUCT(alist:REF accesslist,
                        blocklength:INT,
                        subspacelength:INT,
                        blocktable:REF blocktable);

    USES
      load, store, ¬=, + FROM address;
      accesslist;
      getaddr, setaddr FROM blocktable;

  REALIZATION

    REPRESENTATION
      endspace, nextblock:address;
      spacetable:DYNAMIC ARRAY[0..100] OF
                    STRUCT(aclist:REF accesslist,
                          subspacelnth:INT,
                          btable:REF blocktable);

    IMPLEMENTATION
      CONST blklnth=.....,
            btsize=.....;
      new:
        BEGIN
        WITH spacetable[0] DO
          aclist:=alist;
          subspacelnth:=1;
          btable:=ssbtable
        OD;
        FOR i:=1 TO upb(spacetable,1) DO
          spacetable[i].btable:=NIL
        OD;
        endspace:=nextblock:=bda+blklnth
        END;
      dispose:
        BEGIN END;
```

```
create:
  BEGIN
  VAR ssi, i:INT;
  ssi:=0;
  i:=1;
  WHILE i <= upb(spacetable,1) AND ssi = 0 DO
    IF spacetable[i].btable = NIL THEN ssi:=i FI;
    i+:=1
  OD;
  IF ssi = 0 THEN
    IF extendarray(spacetable,1,100) THEN
      ssi:=i;
      FOR i:=ssi+1 TO upb(spacetable,1) DO
        spacetable[i].btable:=NIL
      OD
    FI
  FI;
  IF ssi ¬= 0 THEN
    WITH spacetable[i] DO
      aclist:=alist;
      subspacelnth:=0;
      sep(btable,btsize)
    OD
  FI;
  create:=ssi
  END;
destroy:
  BEGIN
  VAR block:address;
  WITH spacetable[id] DO
    FOR i:=1 TO subspacelnth DO
      block:=getaddr(btable@,i);
      store(nextblock,block);
      nextblock:=block
    OD;
    dispose(btable);
    btable:=NIL
  OD
  END;
```

```
extend:
  BEGIN
  WITH spacetable[id] DO
    IF subspacelnth = btsize THEN
      extend:=FALSE
    ELSE
      subspacelnth+:=1;
      setaddr(btable@,subspacelnth,nextblock);
      IF nextblock ¬= endspace THEN
        load(nextblock,nextblock)
      ELSE
        nextblock:=endspace:=endspace+blklnth
      FI;
      entend:=TRUE
    FI
  OD
  END;
fetch:
  BEGIN
  WITH spacetable[id] DO
    fetch.alist:=aclist;
    fetch.blocklength:=blklnth;
    fetch.subspacelnth:=subspacelnth;
    fetch.blocktable:=btable
  OD
  END
```

## REFERENCES

[1] Baker, F.T.; "Chief Programmer Team Management of Production Programming"; IBM Systems Journal, vol. 11, no. 1, 1972; pp 56-73

[2] Bensoussan, A., Clingen C.T. and Daley R.C.; "The MULTICS Virtual Memory: Concepts and Design"; CACM, vol. 15, no. 5, May 1972; pp 308-318

[3] Birthwistle, G.M., Dahl, O.J., Myhrhaug, B. and Nygaard, K.; SIMULA begin; Studentlitterature, Auerbach, 1973

[4] Cheatham, T.E. and Townley, J.A.; "A Proposed System for Structured Programming"; Proceedings of Colloque sur la Programmation (April 1974), Lecture Notes in Computer Science (Springer Verlag) vol. 19; pp. 33-40

[5] Clark, B.L. and Horning, J.J.; "The Systems Language for Project Sue"; SIGPLAN Notices, vol. 6, no. 9, Sept 1971; pp. 79-88

[6] Cunningham, R.J. and Pugh, C.G.; "A Language-independent System to Aid in the Development of Structured Programs"; _Software: Practice and Experience_, vol. 6, no. 4, October - December 1976; pp. 487-503

[7] Denning, P.J.; "Virtual Memory"; _Computing Surveys_, vol. 2, no. 3, Sept. 1970; pp 153-189

[8] Deremer, F. and Kron, H.; "Programming-in-the-large versus Programming-in-the-small"; _Proceedings of the International Conference on Reliable Software_, 1975; pp 114-121

[9] Dijkstra, E.W.; "Notes on Structured Programming"; in _Structured Programming_ by Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.W.; Academic Press, London, New York, 1972; pp 1-82

[10] Fischer, A.E. and Fischer, M.J.; "Mode Modules as Representations of Domains"; _Proceedings of the ACM Symposium on Principles of Programming Languages_ (Boston, October 1973); pp 139-143

[11] Good, D.I., London, R.L. and Bledsoe, W.W.; "An Interactive Program Verification System"; _IEEE Transactions on Software Engineering_, vol. 1, no. 1, March 1975; pp. 59-67

[12] Geschke, C.M., Morris, J.H. Jr., Satterthwaite, E.H.; "Early Experience with Mesa"; CACM, vol. 20, no. 8, August 1977; pp 540-553

[13] Henderson, P., Snowdon, R.A., Gorrie, J.D. and King, I.I.; "The TOPD System"; Technical Report Series no. 77, Sept 1975; University of Newcastle upon Tyne

[14] Liskov, B. and Zilles, S.; "Programming with Abstract Data Types"; Proceedings of a Symposium on Very High Level Languages (Santa Monica CA, March 1974); SIGPLAN Notices, vol. 9, no. 4, April 1974; pp 50-59

[15] Liskov, B.; "An Introduction to CLU"; New Directions in Algorithmic Languages 1975, Prepared for IFIP WG 2.1 on ALGOL, Editor Schuman, S.A., 1975; pp. 139-156

[16] London, R.L.; "Proving Programs Correct: Some Techniques and Examples"; BIT, vol. 10, 1970; pp 168-182

[17] Mills, H.D.; "Top Down Programming in Large Systems"; in Debugging Techniques in Large Systems (ed. Rustin, R.); Prentice-Hall, 1971; pp 43-55

[18] Morris, J.H.; "Types are not Sets"; Proceedings of the ACM Symposium on Principles of Programming Languages (Boston, October 1973); pp 120-124

[19] Mraz, A; "An Integrated Tool for the Building of Large Programming Systems"; Proceedings of the IFIP TC2 Working Comittee on the Construction of Quality Software, (1977)

[20] Parnas, D.L.; "Information Distribution Aspects of Design Methodology"; Proceedings of IFIP Congress 1971 (Ljubljana, Yugoslavia, August 1971)

[21] Parnas, D.L.; "On the Criteria to be used in Decomposing Systems into Modules"; CACM, vol. 15, no. 12, December 1972; pp. 1053-1058

[22] White, J.R. and Anderson, R.K.; "Supporting the Structured Development of Complex PL/I Software Systems"; Software: Practice and Experience, vol. 7, no. 2, March - April 1977; pp. 279-293

[23] Wirth, N.; "Program Development by Stepwise Refinement"; CACM, vol. 14, no. 4, April 1971; pp 221-227

[24] Wirth, N.; "Modula: a Language for Modular Multiprogramming"; Software: Practice and Experience, vol. 7, no. 1, 1977; pp 3-35

[25] Wulf, W.A., London, R.L. and Shaw, M.; "Abstraction and Verification in Alphard"; New Directions in Algorithmic Languages 1975, Prepared for IFIP WG 2.1 on ALGOL, Editor Schuman, S.A., 1975; pp. 217-295