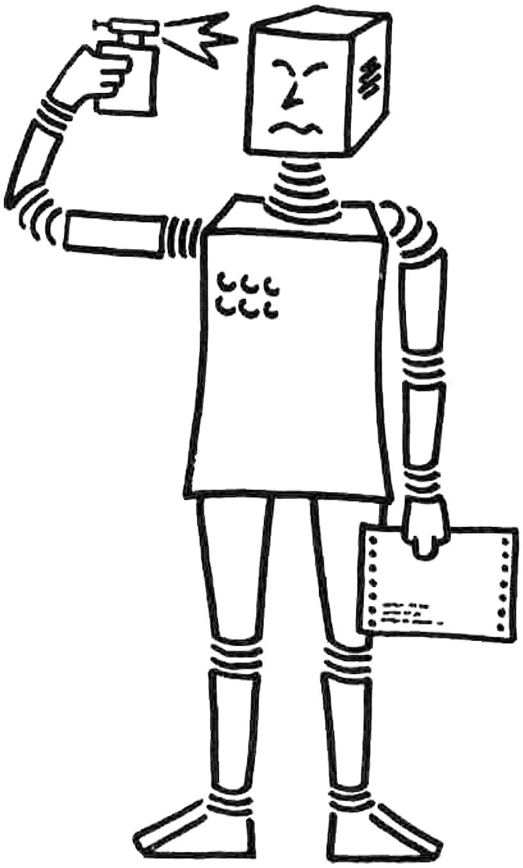
SEKI-PROJEKT

SEKI MEMO

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



ModPascal Report

Walter Olthoff

Memo SEKI-84-09

ModPascal Report

Walter Olthoff

FB Informatik
University of Kaiserslautern
PF 3049
6750 Kaiserslautern
Federal Republic of Germany

Abstract

The object oriented programming language ModPascal and its programming environment are introduced. ModPascal extends Standard Pascal by constructs that have shown usefullness in abstract data type theory: module types, enrichments, instantiations and instantiate types. In fact, ModPascal has been designed as procedural counterpart of a specification language based on abstract data types, and its semantics also employs algebraic structures. ModPascal programs may be edited, compiled and executed by using the ModPascal Programming System that includes a multi-user data base for ModPascal objects.

Keywords: Object oriented programming languages. Parameterization of types. Software engineering environments. Abstract data types.

Contents:

0. Introduction 1

1. Object Oriented Programming in ModPascal 3
   1.1. Module Type Definition 4
   1.2. Enrichment Definition 6
   1.3. Instantiations and Instantiate Types 6

2. Data Management System 9
   2.1. ModPascal and DMS 9
   2.2. DMS Visibility 10
   2.3. Prefixing 10

3. Language Definition 11
   3.1. Overview 11
   3.2. Modules 12
      3.2.1. Syntax 12
      3.2.2. Static Semantics 13
      3.2.3. Variable Declarations 22
      3.2.4. Operation Calls 23
         3.2.4.1. Syntax 24
         3.2.4.2. Static Semantics 24
      3.2.5. Error Operations 26
   3.3. Enrichments 27
      3.3.1. Syntax 27
      3.3.2. Static Semantics 27
   3.4. Instantiations and Instantiate Type Definitions 32
      3.4.1. Syntax of Instantiations 33
      3.4.2. Static Semantics of Instantiations 33
      3.4.3. Syntax of Instantiate Type Definition 37
      3.4.4. Static Semantics of Instantiate Type Definition 37
   3.5. ModPascal Grammar 39

4. Standard Types and Standard Type Generators 46
   4.1. Introduction 46
   4.2. DMS-Structures for Standard Objects 48
      4.2.1. Standard Types 48
      4.2.2. Standard Type Generators 50
   4.3. Mixed Constructs 52

5. Programming in ModPascal 55
   5.1. Main Programs 55
   5.2. ModPascal Programming System 56
   5.3. Precompiling 57

6. Summary 58

7. References 59

0. Introduction

The procedural programming language ModPascal was developed as part of the Integrated Software Development- and Verification System (ISDV-System, [BGGORV 83]). This system employs software engineering techniques along the "verify-while-develop" paradigm: newly introduced structures are verified against formal specifications as soon as possible so that errorneous or inadequate design is detected early before it causes greater damage (=cost of system redesign). This technique is used to link the very first formal specification, the intermediate specification structures and the final ModPascal program by assigning prooftasks (correctness criteria) to all refinement steps. Then, the validity of all prooftasks implies that the ModPascal program meets the requirements imposed by the first formal specification - a proposition that is highly valuable for almost all software developments.

The applied method involves different levels of abstraction and provides concepts and tools for a verifiable transition from abstract to concrete structures. In figure 0-1 a rough overview of the various levels is given together with a also rough classification, and the verification tasks are located.

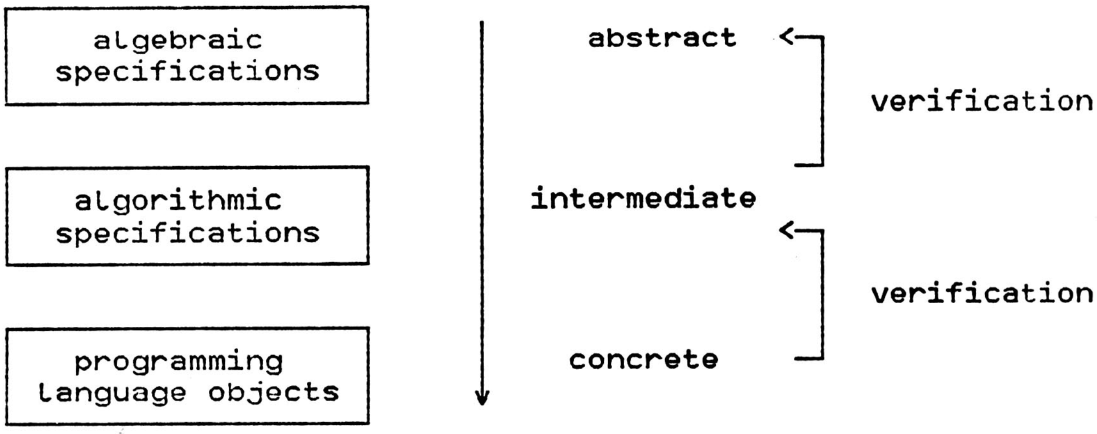


Fig. 0-1: ISDV-System scenario

The formal specifications are given in the applicative specification language ASPIK ([BV 83]) that is strongly based on algebraic specifications ([ADJ 78], [EKP 78]). ASPIK supports incremental, hierarchical software design and offers a number of powerful description features. It is the language of the 'abstract' and 'intermediate' levels of program development in the ISDV-System; the language of the 'concrete' level is ModPascal. As a consequence, both languages offer constructs that are semantically equivalent (e.g. ASPIK specifications - ModPascal modules/enrichments) but exploit the advantages of applicative/procedural languages resp.

During the specification and programming process a number of objects are generated and have to be administrated. This includes as well elementary tasks like storage allocation, link generation or manipulation, checking of object name conventions or access rights, as more pretentious tasks like providing all ISDV subsystems with appropriate input when demanded. This data administration is done by two components of the ISDV-System:

- the file system (FS) for ASPIK objects, and
- the data management system (DMS) for ModPascal and ModPascal related objects.

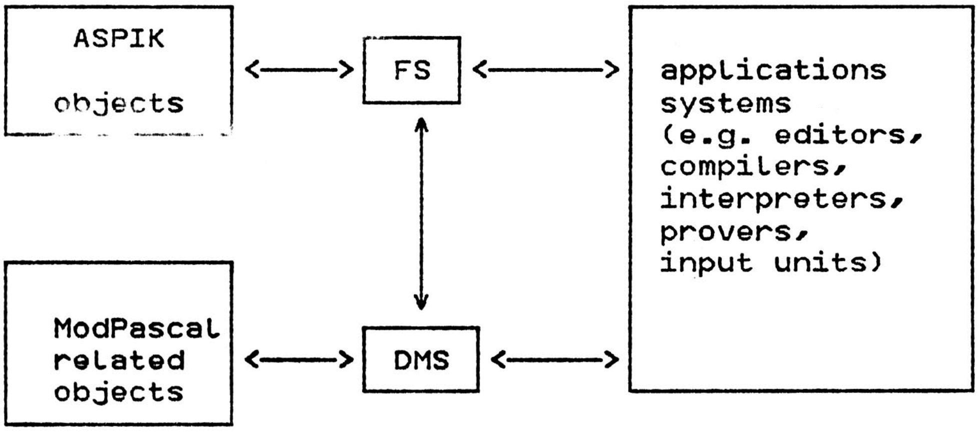


Fig. 2-1: ISDV-System structure

The application systems represent the kernel software. They include input units for interactive, syntax-oriented input of object definitions, editors, a compiler, an interpreter or mechanical proof systems.

One important application system is the ModPascal Programming System (MPPS) which allows to enter and manipulate objects and programs written in ModPascal. MPPS is closely related to DMS, and for this paper it is sufficient to consider only these subsystems of the ISDV-System (sec. 2.1. (DMS) and 5.2. (MPPS)). An overview of the 'concrete' level can be found in [Olt 84c].

In section 1 the main features of ModPascal that are new compared with Standard Pascal are introduced via short examples. Section 2 gives necessary concepts of DMS such that the connections between language and environment becomes visible. The main section 3 deals with the syntax and static semantics of ModPascal but confines itself to the not-Standard Pascal features. Section 4 shows how objects of Standard Pascal are embedded in the ModPascal environment, section 5 treats practical tasks as portability of Standard Pascal programs, and finally section 6 gives an overview on the precompiling concept applied to execute ModPascal programs.

1. Object Oriented Programming in ModPascal

The virtues of object-oriented languages and programming styles have been explored and discussed in many publications. Though it might be necessary to repeat them again and again in order to change inveterate cryptic programming practices, they are suppressed here. Instead, the language ModPascal is introduced, and in doing this some of the advantages of object oriented thinking will inevitably be mentioned.
ModPascal is an extension of Pascal [ISO 7185] in a way that preserves the full set of features of Pascal. The extension has been influenced by two facts:

- In software engineering research algebraic specifications have become widely recognized as a representation independant description method for data types (abstract data types). Algebraic specifications allow modularization and sometimes hierarchization of problem domains and they constitute referential transparency on the specification level (see e.g. [ADJ 78], [EKP 78], [GHM 78], [BV 83]).
- Existing software engineering environments still lack a satisfactory solution to fill the gap between the specification and the final programming languages (e.g. [Sil 80]). Often, it is an incompatibility of language constructs and underlaid semantics that causes the problems.

As a consequence ModPascal has been designed to meet requirements imposed by both theory of algebraic specifications and software engineering environments.
Concerning the latter source of requirements, the reason of its emphasis is, that ModPascal was developed as part of an integrated software development and verfication system [BGGORV 83] that follows the stepwise-refinement and verify-while-develop paradigms. Therefore, many components of the current ModPascal support system are embedded in that environment and reported experience with the language has been gathered there. But it should be pointed out that ModPascal is an imperative, problemoriented programming language (as Pascal is) for wide application, beside its current use in specific software engineering environment.

There are four new kinds of objects that make ModPascal differ from Pascal: modules, enrichments, instantiations and instantiate types. The term 'type' in its usual (Pascal) sense is not applicable to the first three of these constructs since they model more or different information than array, record etc.

To get a feeling of the new structures we will introduce them informally via short examples. The complete definition for modules, enrichments, instantiations and instantiate types will be given in chapters 3,4,5 resp.

1.1. Module Type Definition

Example 1-1: Queue

```
type QUEUE = module
     use TASK;                                        (1)
     public procedure ENTER(T:TASK);                  (2)
            procedure LEAVE;
            function NEXT : TASK;
            function ISEMPTY : BOOLEAN;
            initial EMPTYQUEUE;
     local type T = array [1..100] of TASK;          (3)
           procedure SHIFT(AR:T, I:INTEGER);
           var A:T, PTR:INTEGER;
     local end;

     procedure ENTER;                                 (4)
               var i:INTEGER;
               begin i:=PTR;
                     if i=100 then QUEUE&ERRORPROCEDURE
                     else
                     while i>1 do
                        if T.PRIO>A[i].PRIO
                        then i:=i-1
                     SHIFT(A,i);
                     A[i]:=T;
                     PTR:=PTR+1;
               end;

     procedure LEAVE;
               begin SHIFT(A,0) end;

     procedure SHIFT; (* omitted *)

     function NEXT;
               begin NEXT:= A[1] end;

     function ISEMPTY;
               begin if PTR <> 0
                     then ISEMPTY := FALSE
                     else ISEMPTY := TRUE
               end;

     initial EMPTYQUEUE;
               var i:INTEGER; T:TASK#NEW;
               begin for i:=1 to 100 do
                     A[i] := T;
                  PTR := 0;
               end
modend;
```

Example 1-1 introduces the module type definition. TASK is assumed to be an already defined (module) object with (at least) operations PRIO and NEW. A module type definition mainly consists of four parts: a listing of all used objects of this definition (1), an introduction of all operations that will be tied together by this definition (2), definition of local items that serve to ease programming in part 4 of the module type definition (3), and explicit definitions of the operations introduced in part 2 (4).
This combination of information has condensed out of (at least) two influences of abstract data type theory:

- abstract data types do not describe a specific set but a tuple (set, operations) where the operations are exclusively defined on the set. Moreover, they are the only allowed operations to be performed on 'set'. The programming language construct therefore reflects this fact in introducing only those operations of part 2 as admissable QUEUE-operations.
- an abstract type can be incarnated, i.e. variables of it may be used in abstract programs. Most module concepts do not meet this obvious requirement (eg. Ada packages, Modula modules). They restrict themselves to support solely the combination of procedures, functions, types and variables in a specific syntactic clause, and separate compilation of module header and module body. On the contrary, ModPascal modules allow variable declarations.

From the possibility of using objects it follows immediately, that precautions have to be taken to allow module type definitions which do not explicitly contain declarations of used objects in the surrounding program text. For this purpose a data base has been created that can be referenced and updated by each user of the system. It comprehends as well the Standard Pascal types and type generators as all user defined objects. The data management system administrates all objects in connection with the ModPascal programming system.

1.2. Enrichment Definition

Example 1-2: QUEUE-ENRICHMENT

```
enrichment E-QUEUE use QUEUE is                        (1)
     add TASK                                          (2)
               procedure MERGE(T:TASK);
         QUEUE
               function LENGTH(I:INT):INT;
               procedure SWAP;
     addend;
     procedure MERGE;                                  (3)
               begin ... end;
     function  LENGTH;
               begin ... end;
     procedure SWAP;
               begin ... end;
enrend;
```

Example 1-2 gives a glimpse of enrichment objects. Enrichments are well-known structures in abstract data type (ADT) theory. They are a special case of (algebraic) abstract data type definitions because they do not introduce any new value set but only operations on existing ones. For that reason, enrichments cannot be linked to type definitions since types are defined as set-introducing structures. Consequently, the enrichment-construct has been added to ModPascal. Firstly, it consists of the enrichment identifier and a list of used objectnames (1). To guarantee type uniqueness and type correctness, operations are introduced according to module types (2). In part 2 the procedure MERGE is associated to the module TASK. (TASK is transitively used via QUEUE). This mechanism ensures that whenever the enrichment E-QUEUE is visible the operation MERGE may be performed only on TASK-variables. At last, the full definition of all operations is given (3).

Enrichments may be looked at as a somewhat strange concept. But what they provide is just the above mentioned combination of operations (comp. Ada packages, Modula modules) working on given sets. They establish an extension of the (operation-) name space spanned by the used objects. Additionally, they model a well-known structure of ADT-theory that has proved its adequacy and specification power in many software developments based on abstract data types.

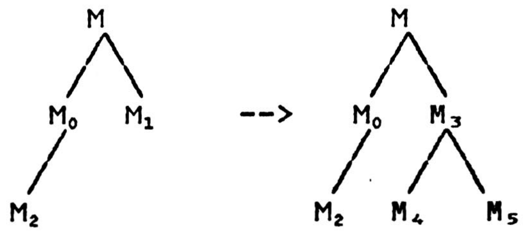
1.3. Instantiations and Instantiate Types

The remaining two structures provided by ModPascal in addition of Pascal are the instantiation construct and the instantiate type definition. In procedural programming languages it is familiar to instantiate operation definitions: each procedure call is an instance of the corresponding procedure definition with formal parameters substituted by information that is elaborated from the actual parameters of the call. The correctness of operation calls is checked on several levels, and depends on criteria such as visibility rules or parameter passing mechanisms that are specific to each language.

But this kind of instantiation is confined to operations. ModPascal allows for a more general concept to instantiate module types. What parts of a module are possible subjects to actualization? To answer, it is helpful to consider the question on standard types as arrays, records etc.. Given a type definition

type A = array [1..10] of INTEGER,

the exchange of the component type INTEGER by for example REAL is desirable, which results in a new type A' with REAL components. Since A 'uses' INTEGER (each array can be seen as module), a first approximation to instantiations of modules is to allow all used objects of a module to be actualized. This includes also the not directly used objects; looking at the tree spanned by the use relation of a module object, instantiations can be visualized as exchanging one arbitrary proper subtree by another.



Recalling the array example the compatibility of INTEGER and REAL (in whatever sense) may be asked for. Since arrays are predefined so are their operations (component selection, assignment). But in arbitrary modules the operations are user-defined and may contain occurrences of operations of the used objects. Now, if some used object is actualized by another, the resulting module would have operation calls of ill defined operations: the defining module is invisible. Therefore, occurences of operations of modules being actualized have to be replaced by occurrences of operations of the actualizing module during the instantiation process. To guarantee that the replacement can be done in all cases, a unique association between all old and new operations has to be given (speaking in terms of abstract data type theory, a signature morphism between the 'formal' parameter object and the 'actual' parameter object has to be stated). The most important constraints on these associations are compatibility of parametertypes of old and new operations and compatibility to other actualizations (one instantiation may actualize two or more objects).

In ModPascal two tasks have been separated in the instantiation of objects:

- In the instantiation definition the programmer has to specify how objects and operations are associated to other objects and operations. Also, he may incorporate already defined instantiations via a use clause. The instantiation definition represents a new object type in ModPascal.

- In the instantiate type definition the replacement of those items associated is actually done. The association has to be specified in a list of instantiation object identifiers, and it is applied to a given module or enrichment object. The result of the instantiate type definition is again a module or enrichment object.

Therefore, an instantiation definition that never occurs in an instantiate type definition, will cause no side effects on generation of objects except generation of itself. In the other case it might be possible that hierarchies of objects have to be generated in evaluating instantiate type definitions.

Example 1-3: QUEUE Instantiation

```
instantiation TASKINT                              (1.1)
              is TASK by INTEGER;                  (1.2)
              operations PRIO = IDENTITY           (1.3)
instend;

type Q' = instantiate QUEUE by TASKINT;            (2)
```

In example 1-3, an instantiation TASKINT is defined and its application inside an instantiate type definition is shown. TASKINT establishes the signature morphism saying 'exchange all occurrences of TASK by INTEGER (1.2) and all occurences of (the only) TASK operation PRIO by the INTEGER operation IDENTITY (1.3)'. It is also possible to specify use clauses and type clauses in instantiation definitions. If other instantiations would have been used they had to be respected by the current signature morphism especially if their sources are hierachically linked objects (see 3.4.2.). If enrichments are involved, a types clause has to be formulated to specify the type mapping for each type occurring in the enrichment.

The generation of a new object is actually done within the evaluation of an instantiate type definition (2). QUEUE, the object to be instantiated, is modified according to the morphism defined in TASKINT. The result is a new object (-hierarchy, possibly) Q' that resembles QUEUE up to TASKINT substitutions. Q' is accessable in the subsequent program text like objects defined by ordinary type definitions.
Standard types like array may also be instantiated using TASKINT. The resulting array type will range over INTEGER

values instead of TASK values.

It should be emphazised that instantiations and instantiate type definitions are more than an extension of parameterization of operations to parameterization of types resp. modules. The set of parameters that have to be actualized in an operation call is fixed by the operation definition and is also mandatory. The set of 'parameters' of a module is only implicitely given: the objects contained in the closure of its use relation. Each subset constitutes a possible set of 'formal parameters' that may be actualized in an instantiation. This concept models a part of the 'parameterization-by-use' feature of ASPIK [BV 83] in ModPascal. For the relations between ASPIK and ModPascal see [RL 85], [SPESY 85] and [OLt 84b].

2. Data Management System (DMS)

2.1. ModPascal and DMS

The DMS represents the object administration system of the concrete level of the ISDV-System (see figure 0-1). It includes the data base for ModPascal objects, access and manipulation operations as well as information retrieval functions. Also the DMS provides the connection to the data manangement system of the abstract level of the ISDV-System, the file system (FS).

To be more precise we anticipate some information section 5 has been dedicated to: programming in ModPascal in the ISDV-System. The ModPascal programming system supports the user by offering tools for editing, compilation and testing of ModPascal programs. Only via MPPS it is possible to enter objects necessary for other ISDV-System components (e.g. proving systems that try to prove correctness criteria in which ASPIK specifications and ModPascal modules are involved).

Passing a program through MPPS involves references to the DMS such that no side effect- and context free relation between input and output can be stated in general. The reason for this is the fact that DMS controls a data base of ModPascal objects that is not fixed. Each reference to DMS in different applications of MPPS may be responded on different states of the data base and such yielding different results.
If, for example, a single module type definition is entered, MPPS checks:

- if there already exists a module definition in the data base with the same name.
- if all used objects of the current definition exist in the data base and if their correctness is guaranteed.
- if no cyclic use-hierarchy is generated by the current module definition.
- if the current module definition is syntactically correct.
- if data base dependent semantic restrictions are respected.

If no check fails, the module definition is added to the data base so that it will be accessable to future module type definitions. Therefore, ModPascal programs having only a few lines of code may enclose a large set of invisible objects of the data base. The advantages of this concept are obvious: once an object definition is entered it can be used in various contexts (e.g. different users) without redefining it again. Also separate compilation is extremely supported since objects communicate only via fixed interface functions so that changes in object-local representations do not affect other objects. Input objects in the sense of MPPS are either single object definitions (modules, enrichments, instantiation types) or 'progs'. The latter contain either a list of object definitions or an ordinary ModPascal program. Each structure of a prog is treated separately by MPPS, and a single object input is simulated: whenever an object definition is recognized completely, it will be installed in the DMS data base before the subsequent part of the prog is examined. This makes semantic checks uniform for prog's and non-prog's. In section 5. the use of prog's is illustrated.

It should be pointed out that MPPS is also applicable without DMS (see sec. 5.), but the current implementation employed in the ISDV-System makes no use of this possibility.

2.2. DMS-Visibility

According to the close relation between MPPS and DMS, accesses of MPPS to DMS-objects have to be carried out along appropriate rules. So it is obvious that objects of the data base should not be accessable by all users, or that protection mechanisms ('read-only access') should qualify the access. This is captured by the concept of DMS-visibility.

Def 2.3.-1 [DMS-visible]

A DMS-object OB is called DMS-visible to user U if either

- a) OB is owned by U
- b) there is another user $\bar{U}$ who ownes OB, and U has (at least) read-access to OB
- c) OB is a system object. ¤

The DMS-visibility is used to define the context sensitive conditions for the correctness of ModPascal objects in sec.3.

2.3. Prefixing

The DMS is drawn up as a multi-user data base that allows cross references to objects of any user. Therefore name conflicts can occur, if for example operations are named equal in different modules that are both visible in some context. In MPPS (see also sec. 5.2.) the following conventions have been set up to avoid ambiguities:

- each object name can be prefixed by a user identification, separated by '&'
- each operation name can be prefixed by either an object name, separated by '&', or by a user identification and an object name, separated by '&'s resp.
- if objects/operations of other users are involved/applied, then the appropriate user identification has to be taken as prefix

  Exception: Standard types as BOOL, INTEGER, REAL, CHAR (see sec. 4.) need not be prefixed by the system standard prefix SYS.

Typical identifier built according to these rules are USER&QUEUE or SYS&TASK&PRIO. For internal purposes of MPPS, all identifiers are extended to their full qualification. If this is not possible in a unique way, an error message is generated.

3. Language Definition

3.1. Overview

ModPascal has been developed to support the hierarchical, verifiable design of programs. Pascal has been choosen as basis for a language extension because many ideas of structured programming have influenced its design, and it supplies the programmer with a set of language constructs that is regarded as standard for a procedural, non-concurrent language. Additionally, Pascal has gained wide recognition in industrial software developments and its choice as target language of the ISDV-System will increase the systems acceptance and applicability concerning industrial problem domains.

ModPascal comprehends Pascal as defined in [ISO 7185]. Therefore all written Pascal programs following that standard will be accepted by MPPS, and already done program work may be incorporated via DMS. Furthermore, ModPascal allows

- module type definitions
- enrichment definitions
- instantiation definitions
- instantiate type definitions
- variable declarations of module type
- invocation of module operations

In Standard Pascal, each program consists of a 'program heading' and a 'block', and each 'block' contains a 'type-definition-part' (the quoted entities represent nonterminals of the Pascal grammar given in [ISO 7185]). In ModPascal programs may also employ module types, enrichments, instantiations and instantiate type definitions so the 'type/enrichment/ instantiation-definition-part' (nonterminal of the ModPascal grammar; see 3.5.) has to be introduced.

Using EBNF the grammatical description is:

```
<type/enrichment/instantiation-definition-part> ::=
              {<type/enrichment/instantiation-definition>}*
<type/enrichment/instantiation-definition> ::=
               <type-definition> | <enrichment-definition>|
                 <instantiation-definition>
```

and refining <type-definition>

```
<structured-type> ::= <module-type> | <instantiate-type> |
                      <unpacked-structured-type> |
                      PACKED <unpacked-structured-type>
```

In the 'type/enrichment/instantiation-definition-part' the object definitions may be given in arbitrary order if they obey Standard Pascal rules as "declaration-before-use" or rules for pointer type definitions. As pointed out in the introduction and section 1. the set of visible objects encloses not only the objects defined in the 'type/enrichment/instantiation-definition-part' but also those being refered to via the DMS. For the correctness of the 'type/enrichment/instantiation- definition-part' it is crucial that the referenced objects of the data base are correct; otherwise a semantic error occurs. The conditions for module type, enrichment and instantiation definitions are given in sec. 3.2., 3.3. and 3.4. The 'variable-declaration-part' is extended to handle module type variable declarations (see sec. 3.2.3.), and those portions of the procedure-and-function-declaration part that concern procedure and function calls are modified to catch invocations of module operations (see sec. 3.2.4.).

The rest of Pascal is left unchanged. The full ModPascal grammar is given in sec. 3.5. For the underlaid semantics of ModPascal see [Olt 84a].

3.2. Modules

3.2.1. Syntax

Modules are introduced as special feature of the Standard Pascal type definition scheme:

```
<type-definition> ::= <identifier> = <type>
<type> ::= <simple-type> | <structured-type> |
                                   <pointer-type>
<structured-type> ::= <unpacked-structured-type> |
        PACKED <unpacked-structured-type> |
        <instantiate-type> | <module-type>
<module-type> ::= MODULE <usepart> <publicpart>
     <localpart> {<modproc/modfuncpart>} <initpart> MODEND
<usepart> ::= USE <uselist>
<uselist> ::= <identifier-list>
<publicpart> ::= PUBLIC <publiclist>
```

```
<publiclist> ::= <publicoperationdcl> |
                 <publicoperationdcl> ; <publiclist>
<publicoperationdcl> ::= <procedureheading> |
                         <functionheading> | <initialheading>
<localpart> ::= LOCAL {<localtypedefpart>}<localvardclpart>
                {<localoperationspart>} LOCALEND ;
<localtypedefpart> ::= <type definition part>
<localvardclpart> ::= <variable-declaration-part>
<localoperationspart> ::= <localoperationlist> ;
<localoperationlist> ::= <localoperationheader> |
             <localoperationheader> ; <localoperationlist>
<localoperationheader> ::= <procedure-heading> |
                           <function-heading>
<modproc/modfuncdclpart> ::= <modproc/modfuncdcllist> ;
<modproc/modfuncdcllist> ::= <modproc-or-modfuncdcl> |
       <modproc-or-modfuncdcl> ;  <modproc/modfuncdcllist>
<modproc-or-modfuncdcl> ::= <modprocdcl> | <modfuncdcl>
<modprocdcl> ::= PROCEDURE <identifier> ; <block>
<modfuncdcl> ::= FUNCTION <identifier> ; <block>
<initpart> ::= <initdcllist> ;
<initdcllist> ::= <initdcl> | <initdcl> ; <initdcllist>
<initdcl> ::= INITIAL <identifier> ; <block>
```

(For the entire ModPascal syntax, see sec. 3.5.)

3.2.2. Static Semantics of Modules

The semantic correctness of a module type definition is determined by the correctness of its constituting parts and its interface correctness:

CM0:

module type definition correct $\Longleftrightarrow$

publicpart correct
$\wedge$ localpart correct
$\wedge$ modproc/modfuncdclpart correct
$\wedge$ initpart correct
$\wedge$ interface correct

CM1: usepart correct
The usepart of a module type definition defines a relation R_u between the current object and the used ones. This relation can be extended to its closure $\bar{R}_u$ that reflects all direct and indirect used objects.

Def 3.2.2.-1 [R_u]

Let M be a module type definition with used objects $U=\{u_1,$

..., u_n}. Then

$R_u(M) := \{(M,u_1), ..., (M,u_n)\}$

denotes the <u>use-relation induced</u> by M. If U is empty, then $R_u(M)$ is the empty set. ¤

<u>Def 3.2.2.-2</u> [$\bar{R}_u$]

Let M be a module type definition with $R_u(M)$ as above. Then $\bar{R}_u(M)$ denotes the least relation with

a) if $(a,b) \in R_u(M)$, then $(a,b) \in \bar{R}_u(M)$

b) if $(a,b) \in \bar{R}_u(M)$, then $R_u(b) \subseteq \bar{R}_u(M)$

$\bar{R}_u(M)$ is called the <u>closure of $R_u(M)$</u> . ¤

Modules may not use themselves; that forces $\bar{R}_u$ to be cyclefree:

<u>Def 3.2.2.-3</u> [cycle, cyclefree]

A <u>cycle</u> C of $\bar{R}_u(M)$ is defined by

a) $C \subseteq \bar{R}_u(M)$

b) Let $|C| = n$, $C = \{(u_1,u_1'), ..., (u_n,u_n')\}$
then $\forall i \in \{1, ..., n-1\}.(u_i' = u_{i+1})$ <u>and</u> $(u_1 = u_n')$

c) C is minimal with respect to b)

$CY(\bar{R}_u(M))$ denotes the set of cycles of $\bar{R}_u(M)$. $\bar{R}_u(M)$ is called <u>cyclefree</u> if $CY(\bar{R}_u(M))$ is empty. ¤

<u>CM1:</u>

usepart correct $\Longleftrightarrow$

all used objects in $\bar{R}_u$ are DMS-visible and correct
$\wedge$ $\bar{R}_u$ is cyclefree

<u>CM2: publicpart correct</u>

The publicpart of a module type definition introduces only the headings (= operation names, formal parameters and formal parameter types) of those operations that are public, i.e. operations that can be invoked from other objects. There are no other items (e.g types, variables) in the public clause.

The publicpart comprises three kinds of headings: procedure-, function- and initial-headings. The definition of at least one initial operation is mandatory since they are necessary for initialization of module variables. Comparing the structure of the operation headings in the publicpart, the main difference lies in the involvation of an implicit formal parameter of the current module type. When called, public operations transform the state of a specific module variable (procedures), extract information of the state of a specific module variable (functions) or assign initial values to a specific variable (initials). In the case of procedures and functions the formal parameter on which the operation is performed is not mentioned in the operation heading. Whenever the operation is called it has to be supplied with an appropriate variable of the current module type which is distinguished from other possible formal parameters of the current module type by syntactic means and which is the specific variable on which the operation is performed (see sec. 3.2.4.). In the case of initials, the treatment is slightly different (see sec. 3.2.3. and 3.2.4.).

The conditions for parameter lists below imply that no function or procedure parameter type may be declared in parameterlists of public operations of a module type definition, although other occurrences (e.g. inside the operation definition blocks, or main program (see sec. 5.1.)) are allowed in ModPascal. The reason is, that interfaces of objects should contain objects too, and not only operations.

<u>CM2:</u>

publicpart correct $\Longleftrightarrow$

all procedure headings correct
$\wedge$ all function headings correct
$\wedge$ all initial headings correct
$\wedge$ at least one initial heading occurs

<u>CM21: procedure heading correct</u>

<u>Def 3.2.2.-4</u> [U(M), U'(M)]

Let M be a module type definition with closure use relation $\bar{R}_u(M)$.
Then

$U(M) := \{a \mid \exists (ob_1, ob_2) \in \bar{R}_u(M).(ob_1 = a \vee ob_2 = a)\}$

is called the <u>set of used objects of M</u> .
Note: $M \in U(M)$.

The set

$U'(M) := \{ob \mid ob \in U(M) \wedge ob \text{ is module type}\}$

is called the <u>set of used modules of M</u>. ∎

U'(M) encloses all those objects which allow declaration of variables.
All parameter types of the procedure heading have to be contained in U'(M).

CM21:

Procedure heading correct $\iff$

the procedure identifier is unique in the module type definition
$\wedge$ all parameter types are contained in U'(M)

CM22 : function heading correct

CM22:

function heading correct $\iff$

the function identifier is unique in the module type definition
$\wedge$ all parameter types are contained in U'(M)
$\wedge$ the result type is contained in U'(M) \ {M}

CM23: initial heading correct

CM23:

initial heading correct $\iff$

the initial identifier is unique in the module type definition
$\wedge$ all parameter types are contained in U(M) \ {M}

CM3: localpart correct

The localpart introduces types, variables, functions and procedures whose scope is restricted to the current module type definition. The only allowed occurrence of local items is inside the operation definitions of the modfunc/modproc- and the initpart. The localvardclpart is mandatory because the underlaid semantics of modules depend strongly on it.

CM3:

localpart correct $\iff$

localtypedefpart correct
$\wedge$ localvardclpart correct

```
     ^ localoperationpart correct
     ^ at least one local variable is declared
```

CM31: localtypedefpart correct

To enforce modular programming in hierarchical structures, the only allowed nesting mechanism for modules is by occurrence in use-clauses. Therefore, module type definitions are the "smallest" modular language constructs in ModPascal (in analogy to algebraic specifications in abstract data type theory) and they do not permit local module type definitions. The spectrum of admissable type definitions in the localtypedefpart is spanned only by Standard Pascal types and type generators (see sec. 4.).

```
CM31:

     localtypedefpart correct ⟺

       no module type definition occurs
     ^ introduced type identifiers are unique in the
       environment of the modul type definition
     ^ all employed types are introduced either in
       the current localtypedefpart or are contained
       in U'(M) \ {M}
```

CM32: localvardclpart correct

For semantical reasons, the local variables are extremly important to assign an appropriate meaning to a module type definition. The variable declarations follow the extended scheme as given in sec. 3.2.5., i.e. declarations of variables of arbitrary explicit type and declaration of variables of implicit, non-module types. The reason to forbid implicit module types is the same as given in CM31.

```
CM32:

     localvardclpart correct ⟺

       all variables are either contained in
       U'(M) \ {M} or are implicit non-module types
     ^ all variables are unique
```

CM33: localoperationpart correct

Similar to the public part, the localoperationpart introduces only operation definitions, but without implicit formal parameters of the current module type, since local operations cannot be called on module variables.

CM33:

localoperationpart correct $\Longleftrightarrow$

all function headings are correct
$\wedge$ all procedure headings are correct

CM331: function heading correct

CM331:

function heading correct $\Longleftrightarrow$

all parameter types and result types are either in U'(M) or are local types
$\wedge$ the function identifier is unique

CM332: procedure heading correct

CM332:

procedure heading correct $\Longleftrightarrow$

all parameter types are either contained in U'(M) or are local types
$\wedge$ the procedure identifier is unique

CM4: modproc/modfuncdclpart correct

Up to now all operations have only been introduced by giving their headings. In the modproc/modfuncdclpart the bodies of just these public and local procedures and functions are defined.

CM4:

modproc/modfuncdclpart correct ⟺

for each public and each local operation there is exactly one body declaration
∧ no other body declaration occurs
∧ all modprocdcl are correct
∧ all modfuncdcl are correct

CM41: modprocdcl correct

A modprocdcl does not repeat the formal parameters and their types. It introduces either the body of a public or a local procedure. In the public case it allows for a Standard Pascal block-structure with some modifications due to the fact that module type variables and operations on them may occur in the procedure body. Public operation calls and occurrences of locally declared items are allowed in a modprocdcl. The distinction on which module variables operations are performed is made by syntactic measures. Standard object operations (see sec.4.) are treated specially. In the local case additionally no invokation of the local operation on a specific module variable is allowed, since locals generally refer to the variable of the original public operation call.

The Pascal correctness conditions are omitted.

CM41:

modprocdcl correct ⟺

global variables are restricted to the local variable set of the current module type definition
∧ each call of a public operation of the current module in prefix notation refers to that module variable on which the original procedure call is performed
∧ all local operations are called in prefix notation
∧ visible items are all public and local items of the current definition without initials, all public items in objects contained in U(M) and all formal parameters
∧ operations of standard types occur in the notation that is given by Standard Pascal
∧ initial operation calls occur only in variable declarations.

CM42: modfuncdcl correct

A modfuncdcl does not repeat the types of the formal parameters and the result type. It introduces either the body of a public or a local function. In the public case it allows for a Standard Pascal block structure with some modifications due to the fact that module type variables and operations on them may occur in the function body. Public operation calls and occurrences of locally declared items are allowed in a modfuncdcl. The distinction on which module variables operations are performed is made by syntactic measures. Standard object operations (see sec.4.) are treated specially. In the local case additionally no invokation of the local operation on a specific module variable is allowed, since locals generally refer to the variable of the original public operation call.

If the function body does modify the local variable values, this causes no side effects to the environment. By the ModPascal semantics this manipulation is kept locally in the body elaboration, and the generated executable code uses appropriate value parameter (see sec. 2. in [RL 85] and [Olt 84a]).

The remaining Pascal correctness conditions are omitted.

CM42:

modfuncdcl correct $\Longleftrightarrow$

global variables are restricted to the local variable set of the current module type definition
$\wedge$ visible items are all public and local items without initials of the current module type definition, public items contained in U(M) \ {M} and all formal parameters
$\wedge$ each call of an operation of the current module type definition in prefix notation refers to that module variable on which the original function call is performed
$\wedge$ all local operations occur in prefix notation
$\wedge$ operations of standard types occur in the notation that is defined by Standard Pascal
$\wedge$ initial operation calls occur only in variable declarations

CM5: initpart correct

The initpart defines operation bodies for just those initial operations whose headings have been given in the public part of the current module.

CM5:

initpart correct ⟺

for each initial operation there is exactly one initdcl
∧ no other initdcl occurs
∧ all initdcls are correct

CM51: initdcl correct

An initdcl does not repeat the formal parameters and their types. It allowes for a Standard Pascal block structure with modifications due to the fact that module type variables and operations on them may occur in the initdcl. Calls of public operations of the current module type definition are not allowed (this includes invokations of other initials of the current module type definition). Local items are as well visible as public items of U(M). Since initials can only be called in variable declarations (see sec.3.2.3.), the variable on which the initial operation is performed is taken from there. Standard object operations are treated specially (see sec. 4.).

The Pascal correctness conditions are omitted.

CM51:

initdcl correct ⟺

global variables are restricted to the local variable set of the current module type definition
∧ visible items are all local items of the current module, all public items in U(M) and all formal parameters
∧ local items occur in prefix notation
∧ operations of Standard Pascal types occur in their usual notation
∧ initial operation calls occur only in variable declarations

CM6: interface correct

The interface correctness of a module encloses conditions on the correct occurrence of object identifier and on the correct call inside operation definitions of operations imported from other modules. For the latter, interface correctness is closely related to correctness of module operation calls (see sec. 3.2.4.).
Because of the semantics of the use clause of a module, the set of admissable imported type identifiers occurring inside operation definitions is restricted to the set U(M) of used objects of M. In the case of used enrichments, this set contains the addpart type identifier (see 3.3.). Only visible operations may be invoked; these are the public and local operations of the current module and the public operations of all used objects. Each call has to satisfy the conventions in number and types of actual parameters that have been imposed by the operations definition.

> CM6:
>
> interface correct $\Longleftrightarrow$
>
> all types occurring without definition in operation bodies have to be contained in the use closure
> $\wedge$ all operation calls occurring without operation definition in operation bodies have to be either defined in the current module or visible public operations of some used object
> $\wedge$ the arities of the calls coincide with the arities of associated definitions

3.2.3. Variable Declarations

The object oriented design of ModPascal considers each type of the language as an autonomous object. This is clear for module type definitions, but it also encloses the Pascal standard types like BOOLEAN, INTEGER etc. or standard type generators like array, record etc. For convenience, in ModPascal the standard objects are treated identically to Pascal. Therefore type definitions, variable declarations and operations on instances of these types have been left unchanged (see sec. 4. for the involvement of standard objects).
The user-defined, non-standard types are the module types (enrichment do not define a new type). Every module is considered to have an internal state, that is exclusively changeable/accessable by the public operations of it. Module incarnations may be generated via variable declarations. To provide a module variable with an initial state, the <module-variable-declaration> construct has been introduced in ModPascal:

```
<general-variable-declaration> ::= <standard-variable-decla-
                         ration> | <module-variable-declaration>
<module-variable-declaration> ::= <identifier-list> :
              <module-identifier> # <initial-operation-call>
```

The <identifier-list> gives a set of variable identifiers that serve to denote the incarnations. The <module-identifier> indicates the module type to be incarnated. The mandatory <initial-operation-call> invokes one of the initial operations of <module-identifier> to assign an initial state to each module variable of <identifier-list>.

CV1:

module variable declaration correct $\Longleftrightarrow$

the module type is DMS-visible and correct
$\wedge$ the initial operation is public in the module type

Initial operations are only allowed in variable declarations - Otherwise, at any point of a program elaboration a module variable might be 'reset' to an initial state - a possibility that is highly unwanted if modules and abstract data types and their relations are considered. As pointed out in the introduction, ModPascal has been designed as part of an integrated software development and verification system, in which abstract data types are used intensively. Therefore ModPascal forbids initial operation calls outside module variable declarations.

Example 3-1
A QUEUE variable (see Example 1-1) may be declared by

var Q:QUEUE # EMPTYQUEUE ;

3.2.4. Operation Calls

To emphasize the object orientedness of ModPascal, special notations and features have been introduced for working with module incarnations :

- extended dot notation for module operations
- left-hand-side module function occurrences in assignments

3.2.4.1. Syntax

```
a)<procedure-statement> ::= <operation-designator>
  <operation-designator> ::= <designator-list> | <operation-
                             designator> . <designator-list>
  <designator-list> ::= <identifier> |
                        <identifier> ( <act-parm-list> )

b)<expression> ::= ... | <factor>
  <factor> ::= ... | <operation-designator>

c)<assignment-statement> ::= <assign-structure> :=
                                                <expression>
  <assign-structure> ::= <compound-variable> |
                         <referenced-variable> |
                         <operation-designator>
```

(For the complete ModPascal grammar, see sec. 3.5.)

3.2.4.2. Static Semantics of Operation Calls

Admissable operation calls in ModPascal may be either Standard Pascal operation calls or module operation calls. The former follow the same syntactic and semantic conditions as in Pascal, while the latter are based on specific notations and semantics.
The general invocation form for module operations is the so-called 'dot notation':

<identifier> . <operation-call>,

where <identifier> has to be a variable of a module type. Additionally the <operation-call> has to contain an invokation of a public operation of the module type of <identifier>.

The effect of the invocation depends on the operations type:

- if the <operation-call> contains a module procedure call, then the module incarnation <identifier> is modified by the body of the procedure.
- if the <operation-call> contains a module function call, then the module incarnation <identifier> is left unchanged; only information of the functions value type is extracted.

This coincides with the view of procedures as 'state transforming operations' and functions as 'state observing operations'.

Example 3-2
From Example 1-1 we get calls

```
    q.ENTER(T)          (1)
    q.LEAVE             (2)
    q.NEXT              (3)
```

(1),(2) modify q, (3) extracts information. ¤

Beside the general invocation form, ModPascal provides an 'extended dot notation'. This covers cases when operation call sequences on the same module incarnation or calls on component modules of a given incarnation have to be programmed. Instead of repeating the relevant module variable or assigning a component module to auxiliary variables before executing the operation of interest, the operation calls may be sequenced by using the 'dot' as delimiter:

<variable-identifier> . <operation-call$_1$> . ... . <operation-call$_n$>

'dot' is left associative; so the sequence is elaborated by first applying <variable-identifier>.<operation-call$_1$>, according to the general invocation forms. The result is passed to <operation-call$_2$>:

- <variable-identifier>, if a procedure had been invoked in <operation-call$_1$>
- a component of <variable-identifier>, if a function had been invoked in <operation-call$_1$>

Then <operation-call$_2$> is elaborated, and so on.
At each step it has to be checked, whether incarnation and operation are compatible. That is, if the operation is public operation in the incarnations module type.
If an operation call in extended dot notation occurs at the left-hand-side of an assignment, then <operation-call$_n$> has to be a function call. Since dot notation is restricted to structured type operations this ensures that always an appropriate substructure of an existing object is evaluated.

<u>Example 3-3</u>

a) q.ENTER(T$_0$);q.ENTER(T$_1$);q.ENTER(T$_2$)
   may be expressed as
   q.ENTER(T$_0$).ENTER(T$_1$).ENTER(T$_2$)
b) T := q.NEXT; ... ; T.PRIO ...
   may be expressed as
   ... q.NEXT.PRIO ...
c) q.ENTER(T$_0$).NEXT := T$_1$
   denotes an assignment with left-hand-side consisting of a dot notated functional expression.

¤

Extended dot notation may also be used for mixed sequences of module procedure and module function calls. But one should be aware, that even if the whole construct evaluates to an expression, it might have had side effects coming from intermediate module procedure calls.

<u>Remark</u>: Extended dot notation also includes access functions on the standard type generators array, record, file, set of Pascal:

```
Let
  A = array [1..10] of INT
  R = record f1:A, f2:A end
  QUEUE(R)  denotes  the  module  of example 1-1,  but
  using R
  q= QUEUE(R) # EMPTYQUEUE;
then
  q.ENTER(r).NEXT.f1[5]
is an admissable expression.
(for standard objects see sec.4.)
```

COC1:

module operation call correct: ⟺

the variable on which the operation is called is of module type
a) [dot notation]:
∧ the operation is public in that module
b) [extended dot notation]:
∧ by elaboration of the sequence of calls from left to right, it holds: each operation is public in the preceding module resp. resulting structure
∧ the operations actual parameter types and their number coincide with the formal parametertypes and their number in the operation definition

3.2.5. Error Operations

A ModPascal module object posseses an internal state that is expressed by the values of its local variables. To specify undefinedness the programmer may use standard error operations that include also exception handling processes like error propagation. An undefined module object is a module object where the value of every local variable is undefined. To each module type definition with type identifier M two operations are implicitely generated:

M&ERRORPROCEDURE
M&ERRORFUNCTION

Both are invokable without parameters, and their object types (procedure, function) are indicated by their names. If an error operation is called the normal evaluation process is interrupted. If an expression contains an error function call, then an error object of the expression type is generated as expression value. If a statement list contains an error procedure call then the value of the module variable on which the error procedure is invoked is set to the undefined module object.

Error operations are public operations of the associated module type definition and obey the same rules as given in 3.2.2.

Example 3-4

a) The public procedure ENTER of example 1-1 contains an application of an error procedure.

b) Let var Q: QUEUE#EMPTYQUEUE denote a variable declaration. Then

    Q := QUEUE&ERRORFUNCTION

is an application of the QUEUE error function.

¤

3.3. Enrichments

Enrichments introduce new operations for previously defined modules. They allow programmers to augment module operation sets in specific environments.

Enrichments have been introduced in ModPascal to allow programmers the specification of objects that solely introduce operations instead of operations and data (as modules do). This corresponds to abstract data type theory where algebraic specifications may have empty sort clauses (see ASPIK ([BV 83]) or [ADJ 78]). Therefore, it is not possible to declare variables of enrichment type since no set of values is defined by the enrichment definition. Enrichments may be used, and they extend the set of public operations for already defined modules. As a consequence, on a variable of a module type the operations of the enrichment may be executed if the enrichment is visible and if the operation is associated to that module (see below).

3.3.1. Syntax

An enrichment definition may occur in the <type/enrichment/instantiation-part>.

```
<enrichment-definition>::= ENRICHMENT <enrichment-identi-
                            fier> USE <object-list> IS
                           <addpart> ADDEND
                           <operation-definition-part> ENREND
<object-list>::= <identifier> | <identifier> , <object-list>
<addpart>::= <addition> | <addition> <addpart>
<addition>::= ADD <identifier> <public-list>
<operation-definition-part>::= <operation-definition> |
          <operation-definition> <operation-definition-list>
<operation-definition>::= <modprocdcl> | modfuncdcl> |
                           <initdcl>
```

Remark: 1) For <public-list>, <modprocdcl>, <modfuncdcl>, <initdcl> refinement, see 3.2.1.
2) For the entire ModPascal syntax, see 3.5.

3.3.2. Static Semantics

The correctness of an enrichment definition is determined by the correctness of its constituting parts and its interface correctness.

CE0:

enrichment definition correct ⟺

  usepart correct
∧ addpart correct
∧ operation-definition-part correct
∧ interface correct

CE1: usepart correct
The usepart of an enrichment definition contains all objects (modules, enrichments) that are used by the current definition. The public items of the used objects may occur in the current definition. The correctness of the usepart depends on similar criteria as useparts of modules (DMS-visibility and cyclefreeness). The definitions 3.2.2.-1,2,3 (R, $\bar{R}_u$, cyclefreeness) take over analogously for the use-clause of enrichments.

CE1:

usepart correct ⟺

  all objects in $\bar{R}_u$ are DMS-visible and correct
∧ $\bar{R}_u$ is cyclefree

CE2: addpart correct
The enrichment definiton does not introduce a new type. Therefore, all operations defined by an enrichment are not implicitely associated with some module type (as it is true for operations defined by a module type definition). On the other hand, there has to be a unique association of each operation to some module type since

- ModPascal is strongly typed; for example, evaluating a functional expression requires information of parameter and value types;
- the enrichment operations will be invoked on specific module incarnations, and to check the correctness of the module operation call it is necessary to know the associated module type operation (see also sec. 3.2.4.).

The addpart of an enrichment provides the means to assign a specific module to a set of newly introduced operations in form of a list 'additions'.
If no addition is specified the enrichment definition is semantically equivalent to its usepart.

CE2:

addpart correct $\Longleftrightarrow$

all additions are correct

CE21: addition correct

An addition extends the set of public operations of a specific module object. This object has to be contained in the use closure of the enrichment definition. In the addition, only headings of operations are given, whereas the operation bodies are defined in the operation definiton part. This separation is analogous to module type definitions.

The nonempty list of function, procedure and initial headings represent the public part of an addition. Again, each heading involves an implicit first formal parameter of that type to which the addition is dedicated. When invoked, public operations transform the state of a specific module variable (procedures), extract information from the state of a specific module variable (functions) or assign initial values to a specific variable (initials). This variable of interest is the actual value of the above mentioned implicit formal parameter, and a special invocation form ('dot-notation') was introduced to emphasize this concept of action (see sec. 3.2.4., for initials 3.2.3.).

CE21:

addition correct $\Longleftrightarrow$

the object of addition is contained in $\bar{R}_u$ and of module type
$\wedge$ the public list is nonempty
$\wedge$ all procedure headings are correct
$\wedge$ all function headings are correct
$\wedge$ all initial headings are correct

CE211: procedure heading correct

For the correctness conditions, definitions 3.2.2.-4 and 3.2.2.-5 take analogously over for an enrichment definition E.

A procedure heading is correct if the procedure identifier is unique as well in the union of additions of E as in the module type definition to which it is associated by the addition. The first condition reflects the fact, that an enrichment operation may be used in the definition of an arbitrary other operation. Even if strong typing of ModPascal would allow for name conflict resolution, overloading of enrichment identifier was disregarded, since abstract data types of the ModPascal environment do not support ambiguous identifier (see [SPESY 85]). Also all parameter types have to be contained in U'(E).

CE211:

procedure heading correct $\Longleftrightarrow$

the procedure identifier is unique in the enrichment and the associated module type definition
^ all parameter types are contained in U'(E)

CE212: function heading correct

CE212:

function heading correct $\Longleftrightarrow$

the function identifier is unique in the enrichment and the associated module type definition
^ all parameter types are contained in U'(E)

CE213: initial heading correct

CE213:

initial heading correct $\Longleftrightarrow$

the initial identifier is unique in the enrichment and the associated module type definition
^ all parameter types are contained in U'(E)
^ no parameter type is the associated module type

CE3: operation definition part correct

In the operation definition part all operations, upto now only introduced by headings, have to be completed in arbitrary order by giving the body definitions. The correctness is derived from the constituting parts of the operation definition part.

CE3:

operation definition part correct $\Longleftrightarrow$

for each introduced operation of the addpart there is exactly one body definition
^ no other body definitions occur
^ all module procedure declarations are correct
^ all module function declarations are correct
^ all initial declarations are correct

CE31: modprocdcl correct

The correctness conditions for modprocdcl inside an enrichment are very similar to those for modprocdcl inside module type definitions. The set of locally declared items is now the set of locally declared items of the addition object, and the set of visible items also includes all operations of the current enrichment.

```
CE31:

     modprocdcl correct ⟺

       CM41 (with obvious substitutions for
       the enrichment case)
     ∧ all operations of the enrichment are visible inside
       the procedure body
```

CE32: modfuncdcl correct

The correctness conditions for modfuncdcl inside an enrichment are very similar to those for modfuncdcl inside module type definitions. The set of locally declared items is now the set of locally declared items of the addition object, and the set of visible items also includes all operations of the current enrichment.

```
CE32:

     modfuncdcl correct ⟺

       CM42 (with obvious substitutions for
       the enrichment case)
     ∧ all operations of the enrichment are visible inside
       the function body
```

CE33: initdcl correct

The correctness conditions for initdcl inside an enrichment are very similar to those for initdcl inside module type definitions. The set of locally declared items is now the set of locally declared items of the addition object, and the set of visible items also includes all operations of the current enrichment, except of those associated to the addition object.

```
CE33:

     initdcl correct ⟺

       CM51 (with obvious substitutions for
       the enrichment case)
     ∧ all operations of other additions of the enrichment
       are visible inside the initial body
```

CE4: interface correct
The interface correctness of an enrichment encloses conditions on the correct occurrence of object identifier, and the correct call inside operation definitions of operations imported from other additions or objects. Interface correctness is closely related to the correctness of module operation calls (see sec. 3.2.4.). According to the semantics of the usepart of an enrichment, the set of admissable imported type identifiers occurring inside operation definitions is restricted to objects of the closure set of the use relation of the enrichment (as generated by definition 3.2.2.-5 for enrichments). This set contains also the addition types of the current enrichment.

Only visible operations may be invoked, that are the public and local operations of the current addition type, all enrichment operations of the current enrichment, and all public operations of the used objects. If some used object is itself an enrichment, then its public operations are all operations defined by it.
Each operation call has to satisfy the conventions in number and type of actual parameters that have been imposed by the operations definition.

CE4:

interface correct $\Longleftrightarrow$

all types occurring without definition in operation bodies have to be contained in the use closure
$\wedge$ all operation calls occurring without definition of operation bodies have to be either defined in the current enrichment or visible public operation of some used object
$\wedge$ the arities of the calls coincide with the arities of associated definitions

3.4. Instantiations and Instantiate Types

The instantiation concept provided by ModPascal may be characterized as a static actualization of arbitrary substructures of object hierarchies by user defined actualizations. The concept is partioned into the instantiation construct and the instantiate type definition. The former establishes a new kind of objects in a procedural programming language: mappings between program identifiers, possibly hierarchically structured. The latter may be seen as type generators like arrays or records, but the generation is directed by user defined instantiations. Result type of an instantiate type definition is always an already existing but possibly modified type.

3.4.1. Syntax of Instantiations

As described in sec. 3.1. instantiations occur in a ModPascal program within a common object definition part (<type/enrichment/ instantiation-definition-part>), where the different kinds of object definitions may be given in arbitrary sequence (as long as declaration-before-use is respected). The syntactical structure of instantiations is :

```
<instantiation-definition> ::= INSTANTIATION <instanti-
             ation-header> <instantiation-body> ENDINST ;
<instantiation-header> ::= <identifier>
<instantiation-body> ::= <usepart> | <actualizationpart> |
                         <usepart> <actualizationpart>
<use-part> ::= USE <object-list> ;
<object-list> ::= <object-identifier> | <object-list> ,
                                        <object-identifier>
<actualizationpart> ::= IS <actualization>
<actualization> ::= <object-actualization> {<type-actualiza-
                    tion>} <operation-actualization>
<object-actualization> ::= <object-actualization-list> ;
<object-actualization-list> ::= <o-actualization-clause> |
     <object-actualization-list> , <o-actualization-clause>
<o-actualization-clause> ::= <object-identifier> BY
                                        <object-identifier>
<type-actualization> ::= TYPES <type-actualization-list>
<type-actualization-list> ::= <t-actualization-clause> |
      <type-actualization-list> , <t-actualization-clause>
<t-actualization-clause> ::= <object-identifier> = <object-
                                               <identifier>
<operation-actualization> ::= OPERATIONS <operation-actu-
                                     alization-list> ;
<operation-actualization-list> ::=<op-actualization-clause>|
                   <operation-actualization-list> ,
                               <op-actualization-clause>
<op-actualization-clause> ::= <operation-identifier> =
                                  <operation-identifier>
```

(For the complete ModPascal syntax, see sec. 3.5.)

3.4.2. Static Semantics of Instantiations

An instantiation definition is correct, if its constituting parts are correct, and if the instantiatian header introduces an object identifier that is unique with respect to DMS-visible object identifier (see Def. 2.3.1.).

The instantiation body contains either a usepart, an actualizationpart or both. In the first case, only a new name is introduced for a collection of existing instantiation objects, in the second case a signature morphism (see below) is stated that is not based on any other object, and the third case represents the hierarchical definition of an instantiation based on existing instantiation objects.

CI1:

instantiation definition correct $\Longleftrightarrow$

at least a usepart or an actualizationpart occurs
$\wedge$ usepart correct
$\wedge$ actualizationpart correct

CI2 : usepart correct
Via the use clause of an instantiation, the programmer may refer to already defined instantiations and is enabled to incorporate them in the current instantiation definition. The correctness of the usepart depends on similar criteria as useparts of modules and enrichments (DMS-visibility and cylefreeness). The definitions 3.2.2.-1,-2,-3 (R, $\bar{R}_u$, cyclefreeness) take over for the use clause of instantiations. In addition, the set of usable objects is restricted to instantiation objects.
Instantiations define a mapping between objects and a mapping between operations. On this the concept of signature morphism is based.

Def 3.4.2.-1 [arity, signature morphism]
Let OB_1, OB_2 be sets of object names (modules, enrichments), and OP_i denote the set of public operations of objects in OB_i, $i \in \{1,2\}$.

1) A mapping $A_i : OP_i \longrightarrow OB_i^*$ (nonempty strings over OB_i) is called arity ($i \in \{1,2\}$).
   If $A(op)=ob_1 ob_2 ... ob_n$, then $ob_1 ... ob_{n-1}$ are called the source of op, and ob_n the target of op.
2) A tuple (f,g) of mappings $f: OB_1 \longrightarrow OB_2$, $g: OP_1 \longrightarrow OP_2$ is called signature morphism, if
   $\forall op \in OP_1$ with $A_1(op)=ob_1 ... ob_n$. $A_2(g(op))=f(ob_1)...f(ob_n)$

¤

Remark : The arity of an operation is the string consisting of all parameter type and value type names. The signature morphism property says, that the mapping between operation names preserves the arity and is compatible with the mapping between objects.

The usepart of an instantiation object generates a closure of instantiation objects. By this also a 'closure' of signature morphisms is generated that itself represents a signature morphism.

Especially, this signature morphism has to respect the hierarchical structure lying on its set of source objects (modules, enrichments). Hierarchy in this context means a closure use relation $\bar{R}_u$ induced by the use relation of an object, and it corresponds to the notation of a directed acyclic graph (see [Olt 84a] for ModPascal hierarchies).

Then the condition for signature morphisms means that whenever an object ob is mapped to another object ob* all its predecessors in the hierarchy will be modified to contain occurrences of ob* solely. This restriction takes also over to arities of operations: if $arity(op)=ob_1 ob ob_2 \mapsto ob_0$ and $ob \mapsto ob^*$ holds, then the modified arity is $ob_1 ob^* ob_2 \mapsto ob_0$, and the conservation of the second arity is examined in the check of the signature morphism property of the instantiation.

(The checking algorithm is described in [RL 85]).

<u>CI2:</u>

usepart correct $\Longleftrightarrow$

all objects are DMS-visible and correct
$\wedge$ $\bar{R}_u$ is cyclefree
$\wedge$ $\forall$ ob $\in$ closure(ob).type(ob)=INST
$\wedge$ $\bar{R}_u$ describes a signature morphism

<u>CI3 : actualizationpart correct</u>

The actualization part consists of object-, type- and operation- actualization. The type actualization may be omitted if all involved objects of the actualization are of module type. Otherwise (if enrichments occur) the type actualization serves to associate the objects occurring in the enrichment to new objects of the actualizing enrichment.

The actualization part together with the usepart has to define a signature morphism. The correctness of the usepart implies this property for the used instantiations, so that only the newly introduced mappings have to be checked for signature morphism property and consistency with used instantiations.
The object-, type- and operation-actualizations are given as associations "id_1 <u>by</u> id_2" (objects) or "$id_1 = id_2$" (types, operations) where the left-hand-side represents the actualized and the right-hand-side the actualizing items.

<u>CI3:</u>

actualization correct $\Longleftrightarrow$

object actualization correct
$\wedge$ type actualization correct
$\wedge$ operation actualization correct
$\wedge$ the signature morphism property holds

<u>CI31 : object actualization correct</u>

The object actualization associates either DMS-visible modules or enrichments. Cross association (of module with enrichment) is not allowed since enrichments do not possess a unique type.

<u>CI31:</u>

object actualization correct $\Longleftrightarrow$

only module or enrichment type objects occur
$\wedge$ objects are associated to objects of the same type
$\wedge$ all occuring objects are DMS-visible

<u>CI32 : type actualization correct</u>
The type actualization is a list of equations involving object names that have to be read as "substitute the left-hand-side of the equation by the right-hand-side". Only module objects may occur since enrichments do not introduce resp. possess an own type (value set).

The set of types to be actualized (the left hand sides of the equations) has to be a subset of the set of types of the actualized enrichments of the object actualization. If the subset is proper, all missing types are assumed to be actualized identically.
The set of actualizing types has to be a subset of the set of types of the actualizing enrichments of the object actualization.

<u>CI32:</u>

type actualization correct $\Longleftrightarrow$

only visible module types occur
$\wedge$ actualized and actualizing types have to be types of corresponding actualized and actualizing enrichments of the object actualization

<u>CI33 : operation actualization correct</u>
In the operation actualization, all public operations of actualized objects occurring, in the object or type actualization have to be associated with public operations of actualizing objects. If an operation is omitted it is assumed to be actualized identically.

The given actualization has to define a signature morphism on operations. Therefore, the arities of the associated operations have to be checked using the object and type actualization of the current instantiation definition and the signature morphism of the used instantiations (for the check algorithm, see [RL 85]).

CI33:

operation actualization correct ⟺

only public operations of the actualized objects are associated with public operations of the actualizing objects
∧ the arities of associated operations obey the signature morphism property

3.4.3. Syntax of Instantiate Type Definition

The instantiate type definition can be given inside the <type/enrichment/instantiation-definition-part> as variant of the type definition (see sec 3.5.) :

```
<instantiate-type> ::= instantiate <old-object-identifier>
                                 by <object-list>
<object-list> ::= <instantiation-identifier> |
            <object-list> , <instantiation-identifier>
```

3.4.4. Static Semantics of Instantiate Type Definition

The instantiate type definition provides the means to apply an instantiation (=signature morphism) to a specific object (including its hierarchy). As result, a new object (with possibly altered hierarchy) is created, in which all modifications are performed that are induced by the instantiation objects of the <object-list>. <object-list> must be nonempty, and each instantiation object has to define a signature morphism. Furthermore, all instantiations together have to describe a signature morphism (this is checked in analogy to the correctness check of the usepart of an instantiation definition; see CI2).

Only module or enrichment types may be instantiated by the instantiate type definition, and the signature morphism will only work correctly, if all its source objects are contained in the hierarchy generated by <old-object-identifier>.

In generating an instantiated hierarchy of objects it might be necessary to create new objects according to the signature morphism requirement. The reason is, that objects might be actualized (= substituted) by the signature morphism that are located somewhere in the hierarchy of <oldobject-identifier>. In general the actualized object has one or more predecessor objects that use it (directly or indirectly). Now the substitution means for each predecessor object that a modified set of used objects is generated, and that possible occurrences of operation calls of the associated operations of the actualized object have to be substituted by associated operations of the actualizing object. Therefore each predecessor object is possibly modified itself. This process is transitive: the predecessors of each predecessor of the actualized object now use a modified object, so that they themselves have to be modified (at least exchange of elements of the use-list), and so on.In a chain reaction the actualization of an object may lead to a completely different (but automatically generatable) hierarchy of objects.

Example 3-5
Consider the module hierarchy

$$\begin{array}{c} M_1 \\ \downarrow \\ M_2 \\ \downarrow \\ M_3 \\ \downarrow \\ M_4 \end{array}$$

and the instantiation

<u>instantiation</u> I <u>is</u> M_4 <u>by</u> M_5 ;
 <u>operations</u> $op_4 = op_5$; <u>instend</u> ;

and the instantiate type definition that employs I:

<u>type</u> $\bar{M}_1$ = <u>instantiate</u> M_1 <u>by</u> I ;

The primary effect of this definition is the substitution of M_4 by M_5 in the M_1 hierarchy. But then M_3 is no longer appropriate since it uses M_4 in its object definition and has possibly occurrences of M_4 operations. So $\bar{M}_3$ is generated (name conventions are implemented in a similar manner) as a copy of M_3 with exchanged use list and substituted operation calls. Now the same argument is applicable to M_2, resulting in $\bar{M}_2$, and finally to M_1 to yield to $\bar{M}_1$ as outcome of the instantiate type definition. ¤

The treatment of implicit generated objects with respect to the data base is dependent on user options. It is possible to include them in the user specific data base (if further use is intended), to hide them in the system managers data base (to keep data bases free from 'technical' objects) or to declare them as temporary objects that together with the explicit generated objects will be deleted at the end of the session if only testing is intended.

The details of the implicit object generation algorithm can be found in [OLt 84a] and [RL 85].

CIT1:

instantiate type definition correct ⟺

<old-object-identifier> is either a module or enrichment type
∧ the elements of <old-object-list> are correct instantiations and they describe a signature morphism
∧ all source objects of the signature morphism are contained in the hierarchy spanned by <old-object-identifier>

3.5. ModPascal Grammar

This section documents the complete grammar of the language. The ModPascal programs are first precompiled into Pascal and then compiled into executable code. The precompiler (see [ECK 84]) has been implemented using a parser generating system for LALR(1) grammars. Therefore, some modifications on the form of production have been done to reach LALR(1) property. The accepted language is not affected by the changes.
The nonterminals <id>, <unsigned-integer>, <unsigned-real>, <string> are not refined; they are recognized by the scanner of the precompiler.

```
<program>                      ::= <program-heading> <block> .
<program-heading>              ::= PROGRAM <identifier> (
                                   <program-parameters> ) ;
<identifier>                   ::= <id>
<program-parameters>           ::= <identifier-list>
<identifier-list>              ::= <identifier> /
                                   <identifier-list> ,
                                   <identifier>
<block>                        ::= <label-declaration-part>
                                   <constant-definition-part>
                                   <type/enrichment/
                                   instantiation-part>
                                   <variable-declaration-part>
                                   <subprogram-declarations>
                                   <statement-part>
<label-declaration-part>       ::= LABEL <lab-list> ;
<lab-list>                     ::= <lab> / <lab-list> , <lab>
<lab>                          ::= <unsigned-integer>
<constant-definition-part>     ::= CONST <constant-
                                   definition-list> / <empty>
<constant-definition-list>     ::= <constant-definition>
                                   / <constant-definition>
                                   <constant-definition-list>
<constant-definition>          ::= <identifier> = <constant> ;
<constant>                     ::= <unsigned-number> / <sign>
                                   <unsigned-number> /
                                   <identifier> / <sign>
                                   <identifier>
<unsigned-number>              ::= <unsigned-integer> /
```

```
                                   <unsigned-real>
<sign>                         ::= + / -
<empty>                        ::=
<type/enrichment/instantiation-part>
                               ::= <type/enrichment/
                                   instantiation-definition> /
                                   <type/enrichment/instantiation-
                                   definition> <type/enrichment/
                                   instantiation-part> / <empty>
<type/enrichment/instantiation-definition>
                               ::= <type-definition-part> /
                                   <enrichment-definition> /
                                   <instantiation-definition>
<type-definition-part>         ::= TYPE <type-definition-list> ;
<type-definition-list>         ::= <type-definition> /
                                   <type-definition-list> ;
                                   <type-definition>
<type-definition>              ::= <identifier> = <ttype>
<ttype>                        ::= <simple-type> /
                                   <structured-type> /
                                   <pointer-type>
<simple-type>                  ::= <scalar-type> / <subrange-type>
                                   / <identifier>
<scalar-type>                  ::= ( <identifier-list> )
<subrange-type>                ::= <constant> .. <constant>
<structured-type>              ::= <unpacked-structured-type> /
                                   <instantiate-type> /
                                   <module-type> / PACKED
                                   <unpacked-structured-type>
<unpacked-structured-type> ::= <array-type> / <record-type> /
                                   <set-type> / <file-type>
<array-type>                   ::= ARRAY [ <index-type-list> ] OF
                                   <component-type>
<index-type-list>              ::= <index-type> / <index-type> ,
                                   <index-type-list>
<index-type>                   ::= <simple-type>
<component-type>               ::= <ttype>
<record-type>                  ::= RECORD <field-list> END
<field-list>                   ::= <fixed-part> / <fixed-part> ;
                                   <variant-part> / <variant-part>
<fixed-part>                   ::= <record-section-list>
<record-section-list>          ::= <record-section> /
                                   <record-section-list> ;
                                   <record-section>
<record-section>               ::= <field-id-list> : <ttype> /
                                   <empty>
<field-id-list>                ::= <identifier-list>
<variant-part>                 ::= CASE <tag-field-identifier> OF
                                   <variant-list>
<tag-field-identifier>         ::= <identifier> : <identifier> /
                                   <identifier>
<variant-list>                 ::= <variant> / <variant-list> ;
                                   <variant>
<variant>                      ::= <case-label-list> : (
                                   <field-list> ) / <empty>
<case-label-list>              ::= <case-label> /
```

```
                                   <case-label-list> ,
                                   <case-label>
<case-label>                   ::= <constant>
<set-type>                     ::= SET OF <base-type>
<base-type>                    ::= <simple-type>
<file-type>                    ::= FILE OF <ttype>
<instantiate-type>             ::= INSTANTIATE
                                   <old-object-identifier> BY
                                   <i_object-list>
<old-object-identifier>        ::= <identifier>
<i_object-list>                ::= <instantiation-identifier> /
                                   <i_object-list> ,
                                   <instantiation-identifier>
<instantiation-identifier>     ::= <identifier>
<module-type>                  ::= MODULE <usepart> <publicpart>
                                   <localpart> [<modproc /
                                   modfuncpart>] <initpart> MODEND
<usepart>                      ::= USE <uselist> ; / <empty>
<uselist>                      ::= <identifier-list> .
<publicpart>                   ::= PUBLIC <publiclist> ; / <empty>
<publiclist>                   ::= <publicproc-func-list>
<publicproc-func-list>         ::= <publicoperationdcl> /
                                   <publicoperationdcl> :
                                   <publiclist> ;
<publicoperationdcl>           ::= <procedure-heading> /
                                   <function-heading> /
                                   <initial-heading>
<procedure-heading>            ::= PROCEDURE <procparms>
<procparms>                    ::= <identifier> / <identifier> (
                                   <formparmsection-list> )
<formparmsection-list>         ::= <formparmsection> /
                                   <formparmsection> ;
                                   <formparmsection-list>
<formparmsection>              ::= <parametergroup> / VAR
                                   <parametergroup> / FUNCTION
                                   <parametergroup> / PROCEDURE
                                   <identifier-list>
<parametergroup>               ::= <identifier-list> :
                                   <identifier>
<function-heading>             ::= FUNCTION <funcparms>
<funcparms>                    ::= <identifier> : <result-type> /
                                   <identifier> (
                                   <formparmsection-list> ) :
                                   <result-type>
<result-type>                  ::= <identifier>
<initial-heading>              ::= INITIAL <identifier>
                                   <initparams>
<initparams>                   ::= <procparams>
<localpart>                    ::= LOCAL <localtypedefpart>
                                   <localvardclpart>
                                   [<localoperationpart>] LOCALEND
                                   ;
<localtypedefpart>             ::= <type-definition-part> /
                                   <empty>
<localvardclpart>              ::= <variable-declaration-part>
<variable-declaration-part>
```

```
                                ::= <vardcl-list>
<vardcl-list>                   ::= <general-variable-declaration>
                                    / <vardcl-list>;
                                    <general-variable-declaration>
<general-variable-declaration>
                                ::= <standard-variable-declaration>
                                    / <module-variable-declaration>
<standard-variable-declaration>
                                ::= <variable-declaration>
<variable-declaration>          ::= <vardcl-kopf> <ttype>
<vardcl-kopf>                   ::= <identifier-list> :
<module-variable-declaration>
                                ::= <identifier-list> :
                                    <module-identifier> #
                                    <initial-operation-call>
<module-identifier>             ::= <identifier>
<initial-operation-call>        ::= <identifier> / <identifier> (
                                    <act-parm-list> )
<act-parm-list>                 ::= <act-parm> / <act-parm-list> ,
                                    <act-parm>
<act-parm>                      ::= <expression> / <variable>
<expression>                    ::= <simple-expression> /
                                    <simple-expression>
                                    <relational-operator>
                                    <simple-expression>
<simple-expresssion>            ::= <term> / <simple-expression>
                                    <adding-operator> <term> /
                                    <sign> <term>
<term>                          ::= <factor> / <term>
                                    <multiplying-operator> <factor>
<factor>                        ::= <variable> /
                                    <unsigned-constant> /
                                    <function-designator-part> /
                                    <sett> / ( <expression> ) / NOT
                                    <factor>
<variable>                      ::= <component-variable> /
                                    <referenced-variable> /
                                    <identifier>
<component-variable>            ::= <indexed-variable> /
                                    <field-designator>
<indexed-variable>              ::= <array-variable> [
                                    <expression-list> ]
<array-variable>                ::= <variable>
<expression-list>               ::= <expression> /
                                    <expression-list> ,
                                    <expression>
<field-designator>              ::= <component-variable> .
                                    <identifier> / <identifier> .
                                    <identifier> /
                                    <referenced-variable> .
                                    <identifier>
<referenced-variable>           ::= <pointer-variable> @
<pointer-variable>              ::= <variable>
<unsigned-constant>             ::= <unsigned-number> / <string> /
                                    <identifier> / NIL
<function-designator-part> ::= <operation-designator>
```

```
<operation-designator>       ::= <designator-list> /
                                 <operation-designator> .
                                 <designator-list>
<designator-list>            ::= <identifier> / <act-list>
<act-list>                   ::= <identifier> ( <act-parm-list>
                                 )
<sett>                       ::= [ <element-list> ]
<element-list>               ::= <elementlist> / <empty>
<elementlist>                ::= <element> / <elementlist> ,
                                 <element>
<element>                    ::= <expression> / <expression> ..
                                 <expression>
<multplying-operator>        ::= * / / / DIV / MOD / AND
<adding-operator>            ::= + / - / OR
<relational-operator>        ::= <> / = / < / > / <= / >= / IN
<localoperationpart>         ::= <localoperationlist> ;
<localoperationlist>         ::= <localoperationheader> /
                                 <localoperationheader> ;
                                 <localoperationlist>
<localoperationheader>       ::= <procedure-heading> /
                                 <function-heading>
<modproc/modfuncpart>        ::= <modproc/modfuncdcllist> ;
<modproc/modfuncdcllist>     ::= <modproc-or-modfuncdcl> /
                                 <modproc/modfuncdcllist> ,
                                 <modproc-or-modfuncdcl>
<modproc-or-modfuncdcl>      ::= <modprocdcl> / <modfuncdcl>
<modprocdcl>                 ::= PROCEDURE <identifier> ;
                                 <block>
<modfuncdcl>                 ::= FUNCTION <identifier> ; <block>
<initpart>                   ::= <initdcllist> ;
<initdcllist>                ::= <initdcl> / <initdcllist> ;
                                 <initdcl>
<initdcl>                    ::= INITIAL <identifier> ; <block>
<pointertype>                ::= @ <identifier>
<enrichment-definition>      ::= ENRICHMENT <enrichment-identi-
                                 fier> USE <e-object-list> IS
                                 <addpart> ADDEND
                                 <operation-definition-part>
                                 ENREND
<enrichment-identifier>      ::= <identifier>
<e-object-list>              ::= <enrichment-identifier> |
                                 <enrichment-identifier>,
                                 <e-object-list>
<addpart>                    ::= <addition> | <addition>
                                 <addpart>
<addition>                   ::= ADD <identifier> <publiclist>
<operation-definition-part>
                             ::= <operation-definition> |
                                 <operation-definition>
                                 <operation-definition-part>
<operation-definition>       ::= <modprocdcl> | <modfuncdcl> |
                                 <initdcl>
<instantiation-definition>   ::= INSTANTIATION <instanti-
                                 ation-header>
                                 <instantiation-body> ENDINST ;
<instantiation-header>       ::= <identifier>
```

```
<instantiation-body>            ::= <usepart> | <actualizationpart>
                                    | <usepart> <actualizationpart>
<actualizationpart>             ::= IS <actualization>
<actualization>                 ::= <object-actualization>
                                    {<type-actualization>}
                                    <operation-actualization>
<object-actualization>          ::= <object-actualization-list> ;
<object-actualization-list>
                                ::= <o-actualization-clause> |
                                    <object-actualization-list> ,
                                    <o-actualization-clause>
<o-actualization-clause>        ::= <object-identifier> BY
                                    <object-identifier>
<object-identifier>             ::= <identifier>
<type-actualization>            ::= TYPES <type-actualization-list>
<type-actualization-list>       ::= <t-actualization-clause> |
                                    <type-actualization-list> ,
                                    <t-actualization-clause>
<t-actualization-clause>        ::= <object-identifier> = <object-
                                    <identifier>
<operation-actualization>       ::= OPERATIONS <operation-actu-
                                    alization-list> ;
<operation-actualization-list>
                                ::= <op-actualization-clause>|
                                    <operation-actualization-list>
                                    , <op-actualization-clause>
<op-actualization-clause>       ::= <operation-identifier> =
                                    <operation-identifier>
<operation-identifier>          ::= <identifier>
<subprogram-declarations>       ::= <sub-declaration-part>
                                    <subprogram-declarations> /
                                    <empty>
<sub-declaration-part>          ::= <sub-declaration> ;
<sub-declaration>               ::= <proc-declaration> /
                                    <func-declaration>
<proc-declaration>              ::= <procedure-heading> ; <block>
<func-declaration>              ::= <function-heading> ; <block>
<statementpart>                 ::= <compound-statement> / <empty>
<compound-statement>            ::= BEGIN <statement-sequence> END
<statement-sequence>            ::= <statement> /
                                    <statement-sequence> ;
                                    <statement>
<statement>                     ::= <unlabelled-statement> / <lab>
                                    : <unlabelled-statement>
<unlabelled-statement>          ::= <simple-statement> /
                                    <structured-statement>
<simple-statement>              ::= <assignment-statement> /
                                    <procedure-statement> /
                                    <goto-statement> / <empty>
<assignment-statement>          ::= <assign-structure> :=
                                    <expression>
<assign-structure>              ::= <component-variable> /
                                    <referenced-variable> /
                                    <operation-designator>
<procedure-statement>           ::= <operation-designator>
<goto-statement>                ::= GOTO <lab>
```

```
<structured-statement>      ::= <compound-statement> /
                                <conditional-statement> /
                                <repetetive-statement> /
                                <with-statement>
<conditional-statement>     ::= <if-statement> /
                                <case-statement>
<if-statement>              ::= IF <expression> THEN
                                <statement> / IF <expression>
                                THEN <statement> ELSE
                                <statement>
<case-statement>            ::= CASE <expression> OF
                                <case-list> END
<case-list>                 ::= <case-list-element> /
                                <case-list-element> ;
                                <case-list>
<case-list-element>         ::= <case-label-list> : <statement>
                                / <empty>
<repetetive-statement>      ::= <while-statement> /
                                <repeat-statement> /
                                <for-statement>
<while-statement>           ::= WHILE <expression> DO
                                <statement>
<repeat-statement>          ::= REPEAT <statement-list> UNTIL
                                <expression>
<statement-list>            ::= <statement> / <statement-list>
                                ; <statement>
<for-statement>             ::= FOR <control-variable> :=
                                <for-list> DO <statement>
<control-variable>          ::= <identifier>
<for-list>                  ::= <initial-value> TO
                                <final-value> / <initial-value>
                                DOWNTO <final-value>
<initial-value>             ::= <expression>
<final-value>               ::= <expression>
<with-statement>            ::= WITH <record-variable-list> DO
                                <statement>
<record-variable-list>      ::= <record-variable> /
                                <record-variable-list> ;
                                <record-variable>
<record-variable>           ::= <identifier>
```

4. Standard Types and Standard Type Generators

4.1. Introduction

Since ModPascal extends Pascal the question arises how to treat the Pascal standard types (BOOLEAN, INTEGER, REAL, CHAR) and type generators (ARRAY, RECORD, FILE, SET, POINTER, ENUMERATION, SUBRANGE) in an object oriented environment. Should they be redefined to fit into the module definition frame, with the effect of redefining also familiar functions and notations?

From a theoretical point of view there is no difference between standard objects and non standard objects as modules or enrichments. To each of them the same semantic structure (algebra) is assigned. This becomes clear if one realizes that for example the type identifier INTEGER in Pascal/ModPascal does not only denote the set {...,-1,0,1,...} but also provides the appropriate arguments for the '+' operator. If coercions are disregarded (although fitting for society, they obscure programs similar to goto's), then '+' is only applicable to INTEGER values - a fact that is at least sufficient to group the set and the operator more closely, for example in an algebra. This is true also for '-', 'div', '*', 'faculty' etc., and also Pascal/ModPascal BOOLEAN (operators: 'AND', 'OR', 'NOT' etc.), REAL ('+', '-',..., 'sin', 'cos' etc.) and CHAR (implementation dependent, but including subrange operations as predecessor, successor, '<', '>=' etc.) describe semantically the same type of structure as module type definitions, that is an algebra.

The standard type generators differ only slightly. For instance, an 'array [1..10] of INTEGER' describes as value set 10-tuples of INTEGER values, and operations only applicable to arrays are assignment (':=') and selection ('_[_]'). Records, files, sets, with restrictions pointer, enumeration and subrange types can all together be associated with algebras, so that a module type definition would not be senseless.

But looking at the definition scheme for modules (sec. 3.) one has to provide a set of local variables that are used to describe the value set of the associated algebra (cartesian product of local variable types). In general, this is impossible even unnatural in the case of standard type generators. For example the above mentioned array object could be defined by a ModPascal module type definition by using ten local variables of type INTEGER. But for standard types even this clumsy way of definition fails. There is no way to represent, for example, INTEGER values in a module type definition by local variables of other type than INTEGER - and that means being circular. The reason is that standard type generators possess at least one component or parameter type and the representation in a module type definition may easily be taken as an appropriate vector of component type variables. In this view, standard types are basic and are not definable by module type definitions.

Even if this fact is sufficient enough to treat standard objects of ModPascal apart from modules, another difficulty should be mentioned. All objects administrated by the DMS form some kind of hierarchy, since this is highly valuable for incremental software development and verification. In general, the hierarchy is built upon a use relation. Being hierarchical implies: no cycles occur. But looking at INTEGER and BOOLEAN of ModPascal, the former type encloses predicate operations as '<' (less) which evaluate to a boolean value and therefore 'INTEGER uses BOOLEAN'. On the other hand, in Pascal BOOLEAN is considered as instance of a two value enumeration type (false,true) ([ISO 7185]), and it involves all operations normally found for enumeration types. So there is an ord(er) operation that evaluate to an INTEGER number indicating the position of an item in the defining sequence (e.g. ord(true) = 2). But this means 'BOOLEAN uses INTEGER' and a cycle is introduced. The solution of this problem with respect to the DMS is described in sec. 4.2.

From all this it comes out to treat the standard types and the standard type generators in two levels:

- the language definition introduces the standard objects of ModPascal identically to those of Pascal. No module type definition is employed, so that the Pascal type set is a proper subset of the ModPascal type set. This guarantees portability of programs and avoids irritation of programmers that are confronted with artificial definitions of familiar types;
- the ModPascal environment (DMS and MPPS) installs predefined objects for the standard types and predefined object generators for standard type generators. These objects represent the semantic structure of types and types defined by the generators, and they contain for example lists of operations and their functionalities associated to the type. Also special objects are predefined that enable decomposition of cyclic structures.

This design has consequences as well for the semantics of ModPascal programs as for the algorithms of the ModPascal precompiler that check correctness. Firstly, the semantics of a ModPascal object becomes context sensitive in that sense that the current state of the data base of DMS is essential for semantic computations. Deleting or manipulating objects might have side effects on the correctness of some other objects. Secondly the unchanged syntax of standard objects together with a new module-like semantics induces a variety of syntactic and semantic problems that occur by combining standard objects with non standard objects (e.g. enrichments of INTEGER). Both consequences will be tackled in the following sections.

4.2. DMS-Structures for Standard Objects

4.2.1 Standard Types

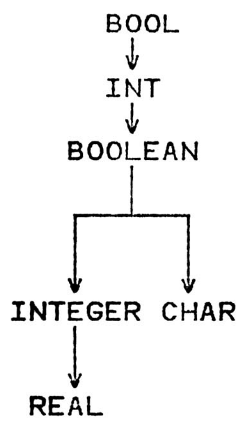
The set of standard types of ModPascal comprises BOOLEAN, INTEGER, REAL and CHAR (= Pascal standard types). Each type has associated a number of operations that are either explicitly characterized (in [ISO 7185]) as belonging to the type (e.g. the 'and' operator belongs to BOOLEAN) or are implicitly derivable from overloaded general operators (e.g. the ':=' (assignment) operator or the '=' (equality) operator, that may be associated with each standard type). This association is fixed by the language definition so that user defined programs cannot modify standard type structures.

The DMS destinguishes between its users: among them there is one - the system manager SYS - which has unrestricted access to all objects, contrary to the limitations that are imposed on ordinary users. SYS ownes also objects, but most of them possess two important characteristics: they are viewed as fixed, and all other users do have read-access to them, so that they can incorporate SYS-objects arbitrarily. This 'general library property' of the SYS-object set is best suited to include ModPascal standard types. Therefore they are defined as SYS-objects and accessable to all users (as Pascal standard types are available in each Pascal program).

The problem of circularity in the ModPascal standard type hierarchy (see 4.1.) is solved by introducing two new objects:

- BOOL, which is a restriction of BOOLEAN to its essential operations 'TRUE' and 'FALSE', and
- INT, which is a restriction of INTEGER to its essential operations 'ZERO', 'PRED' and 'SUCC'.

Only the most necessary ingredients of BOOLEAN or INTEGER, without which the type is undefinable, were choosen for BOOL and INT. All additional operations - including the trouble making 'ord' - are defined in 'higher' objects (see [RL 85]). The resulting hierarchy of standard types is as follows:



BOOL, INT, REAL and CHAR are modules, while BOOLEAN and

INTEGER are enrichments (of BOOL, INT resp.). Since BOOL and BOOLEAN are intended to work on the same data set (of boolean values = {true, false}) both cannot be modules. A module type definition introduces its own value set, that is by definition the source on which module operations are exclusivly invocable. In the case of BOOL and BOOLEAN this would lead to incompatibility of their operations - in contrary to the intention. Therefore BOOLEAN (and also INTEGER) was introduced as enrichment which guarentees that the 'new' operations work on the same data set (see 3.3.).

This hierarchical structure of standard objects is implemented in the DMS. Each object is defined by a set of flags and properties depending on the object type (module, enrichment, standard etc.). For standard types, the following items are defined:

```
FLAGS: SYNTAX    (indicates syntactical correctness; trivially
                 true)
       INTERFACE (context sensitive conditions; true)
       USED      (existance of using objects; true except
                 BOOL)
       STANDARD  (qualifier; true)
PROPERTIES: RIGHTS (access rights)
            USE    (used objects)
            USED   (using objects)
            PUBLIC (list of public operations and
                   functionalities)
            TYPE   (either MOD(ule) or ENR(ichment))
```

(This list of properties is incomplete; see [RL 85]).

These object definitions are intended to guarantee consistency of sets of user defined objects (e.g. modules). If standard types are referenced by some element of the set, the ModPascal precompiler will check its existence in the data base (as for every used object) and then will perform correctness checks using the information provided by object flag and object property values.

Remarks:

1) It should be pointed out that concerning standard types Pascal compiler checks coincide with some ModPascal precompiler checks; but algorithms become uniform for all objects and errors are deleted as early as possible.
   The artificial objects BOOL and INT are not ModPascal standard types so that type definitions or variable declarations may not incorporate them. Their only application is their existance in the DMS to allow cyclefree hierarchies.
2) In general, enrichment identifiers may not be used as type identifiers (see sec. 3.3.). In the case of INTEGER and BOOLEAN there is an exception. They may be used as type identifiers, and the ModPascal precompiler will recognize it appropriately. The reasons for this inconsistency are compatibility with Standard Pascal, convenience by familiar

structures, and invisibility of the basing modules BOOL and INT in ModPascal.

4.2.2. Standard Type Generators

The set of standard type generators of ModPascal comprises array, record, file, set, pointer, enumeration, and subrange types (=Pascal standard type generators). In opposition to standard types they do not have an initial meaning in the language definition, because essential information (e.g. array size, component type) is missing. This information must be provided in an explicit type definition by the programmer, so that the semantics of a standard type generator will become computable.

Despite of that fact, there are fixed structures for each type generator. For example, arrays do always come with a selection operator '[_]' or pointers with a dereferencing operator. These sets of operation frames (since functionalities are not fixed) are associated to each standard type generator and they are complete in that sense that the actualization done in a type definition does not add or delete operations to or from them.

Therefore, the DMS makes object patterns available for each standard type generator. If a type definition occurs, the parametric parts are actualized, and the resulting well-defined object is entered into the data base. All object patterns represent module type objects since, on the semantic level, (actualized) arrays, records, etc. do not differ from module type definitions, so that also algebras are assigned to them.

The standard type generators could also be ordered in a 'hierarchy' over the standard types, but the hierarchical relation used below is constructed solely for pedagogical purposes (since type generators are not types, and thus cannot be mixed with standard types).

```
                    STANDARD TYPES
                    /    |    |   \
ENUMERATION_TYPES RECORD FILE POINTER
       |
 SUBRANGE_TYPES
    /      \
 ARRAY     SET
```

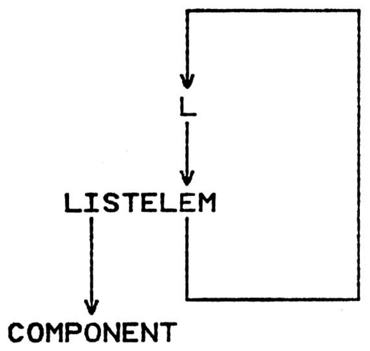
(possible connections between the standard type generators are omitted).

The language definition of Pascal does not include cycles in the hierarchy of standard type generators, so that no artificial objects have to be introduced (see 4.2.1.). But it should be emphasized that objects generated by standard object generators may very well induce cycles. Due to the fact that the 'declaration-before-use' paradigm is ignored in Pascal pointer type definitions, a cycle is easy constructable:

Example 4-1

```
    type L = ↑ LISTELEM
    type LISTELEM = record  f1: COMPONENT;
                            f2: L end;
```

which yields in the hierarchy



¤

Even if this problem is solved by the Pascal compiler, ModPascal does not allow this type of definition. The reason is the intended use of the language as a counterpart of the algebraic specification language ASPIK (see sec. 0.) inside a software development and verification system, and cyclefreeness of object trees is one of the basing features there.

In opposite to standard types, the objects generated by standard type generators are not objects owned by the system manager SYS. They are assigned to the individual user who has entered the generating type definition that also contains the object identifier used by DMS. The object description is similar to standard types and modules, and it comprises (among others) the following flags and properties:

FLAGS: SYNTAX (indicates syntactical correctness; trivially true)
INTERFACE (context sensitive conditions; true)
USED (existance of using objects; true except BOOL)
STANDARD (qualifier; true)

PROPERTIES: RIGHTS (access rights)
USE (used objects)
USED (using objects)
PUBLIC (list of public operations and functionalities)
TYPE (either MOD(ule) or ENR(ichment))

(This list of properties is incomplete; see [RL 85]).

Once an object generated in this way is entered in the data base, it may be manipulated in the same way as all ModPascal objects. The great advantage of this uniform treatment is, that the notational differences between module and non-module type definitions, induced by portability and convenience considerations, are wiped away. The data base therefore reflects the semantics of a set of object definitions more clearly.

Remarks: 1) Sometimes it might be convenient to incorporate objects generated by object generators only temporarily. There are modes in MPPS that enable this .

2) The checks performed by the ModPascal precompiler on standard type generators do only involve the user supplied parts; for the rest correctness is assumed.

4.3. Mixed Constructs

The inclusion of standard objects in ModPascal was done without modification of the Pascal syntax although the semantics were changed. This way is straight forward but there are a number of problems arising in structures that contain as well standard Pascal constructs as ModPascal-specific constructs. Often a solution is possible if syntactic requirements are relaxed or if additional checking is performed by the precompiler.

The correctness checks for mixed constructs are performed in parallel to those of sec. 3.2. and 3.3. So the environmental information is assumed in the following, where the possible situations of mixing are depicted:

A) Object Definitions
In standard and non-standard object definitions types and enrichments may occur along three rules:

A1) Objects generated by standard object generators may reference user defined module types in their definition scheme.
A2) Module type definitions may reference standard objects, user defined modules or user defined enrichments in their useclause.
A3) Enrichment definitions may reference standard objects or user defined modules in their addparts and standard objects, user defined modules or enrichments in their use clause.

These conventions are based on the underlaid semantics for ModPascal. There, components of standard objects must have their own value set (which excludes enrichments in A1), and adding of operations may only be done for objects with value set (which excludes enrichments in A3).

By this, correctness conditions CM1 and CE1 are extended.
[Below, the object generation by a standard object generator

is refered to as structured type definition; see ModPascal grammar, sec. 3.5.]

CMIX1:

structured type definition correct ⟺

  type is correct
∧ used types are user defined module types

CMIX2:

module type definition correct ⟺

  CM0 holds
∧ used objects are standard objects, user defined module types and enrichments

CMIX3:

enrichment definition correct ⟺

  CE0 holds
∧ addparts may be build upon standard objects or user defined modules
∧ used objects are standard objects, user defined modules and enrichments

B) Operation Definitions
Standard object operations are usually predefined and fixed. Only by enrichment of standard objects one is able to define new standard object operations. Then it holds, that
- only functions may be defined for standard objects
- the function body does not contain global variables
- the functionality does not include an implicit formal parameter.

This leads to modification of CE21 and CE32 for the standard object case:

CMIX4:

addition correct ⟺

  the object of addition is contained in $\bar{R}_u$ and of module type
∧ the public list is non-empty
∧ all function headings are correct
∧ no procedure or initial occur
∧ no implicit parameter is introduced

<u>CMIX5:</u>

modfuncdcl correct $\Longleftrightarrow$

  CE32 holds
$\wedge$ the function body does not contain global variables

C) Operation Calls
Usually module operations are called in 'dot-notation' (see sec. 3.2.4.) except occurrences in operation definitions of their associated module type (see CM41/CM42 in sec. 3.2.2.).
This is not true for standard object operation calls. They are invoked in their usual prefix, infix or mixfix notation. Since their association to standard objects is fixed and unique, the precompiler is able to recognize them properly.
Calls of standard object operations that are defined by enrichments always use prefix notation.

<u>CMIX6:</u>

standard operation call correct $\Longleftrightarrow$

  Pascal conventions are respected

<u>CMIX7:</u>

enrichment defined standard operation call
correct $\Longleftrightarrow$

  the defining enrichment is DMS-visible
$\wedge$ prefix notation is used

D) Variable Declarations
Variables of standard object types are not initialized.

E) Prefixing
The standard types need not be prefixed by SYS. Objects generated by standard object generators obey the same prefixing rules than user defined modules or enrichments (see sec. 2.2.).

5. Programming in ModPascal

5.1. Main Programs

Up to now the main emphasis has been put on single objects and object types of ModPascal, and how to include them in a data base of a special purpose programming environment. Object oriented programming is encouraged by the language as much as possible but conventional styles have to be covered anyway. To guarantee portability of already written Pascal code provisions were taken to incorporate programs that are not object oriented.

In MPPS, the vehicle for this are objects of type 'prog'. If prog objects are entered, the system expects as input ordinary programs consisting of label-, type-, variable-, function- and/or procedure declarations and a statement part (block).

Example 5-1:

```
program TEST (input,output);
type M1 = module ... ;
     A  = array ... ;
instantiation I use ...;
enrichment E use ...;
type M2 = instantiate ...;
function ... ;
var ... ;
begin
...
end.
```

Example 5-1 shows a prog object. ¤

If object definitions as type definitions, enrichment definitions or instantiation definitions are recognized they are subsequently submitted to the data base (if no contradicting user option is active). This means that no explicit connection between the object and its defining prog is saved, and it simulates the behaviour of single object input.

The remaining variable declaration, operation definition and statement part are connected to the prog object making it similar to module objects. In this view the entire statement part is seen as a special operation of the prog object, an implicit declared 'statement procedure'. If nested block structures occur this separation between object and non-object definition is performed for each block.

If errors occur, the user can correct them by entering the ModPascal editor (currently a standard file editor). Otherwise the precompiled object (now connected to a Pascal program) can be passed to the Pascal compiler which checks Pascal relevant semantics and generates executable code. At last, in a testing environment programs written in ModPascal may be excecuted.

From this it might be directly suggested to impose a tree structure on progs: hierarchical relations to other progs, interface operations and internal states are easily derivable in the parsing process, and the 'statement procedure' generation is just a naming problem. If some prog fails to meet requirements the user could be advised to correct it.

But this solution is rejected in the MPPS. It would have reversed the goals of an object oriented programming language since it enforces a kind of hidden modularization that is only visible to the system. All disadvantages of conventional programming would take over when modifying or exchanging prog objects of this style.

Instead progs are treated in the mixed fashion as described above. Those parts which can easily be transformed to object oriented formalisms are grasped and included while other parts are disregarded from further use. In the consequence this leads programmers to relinquish prog objects in their work and to use that object definition patterns that are offered by ModPascal. Application of object oriented techniques then exhibits the 'statement procedure' as a public operation of a user defined module object, and conventional programs are expressed by a module (enrichment) object hierarchy.

5.2. The ModPascal Programming System (MPPS)

The MPPS provides the user interface for the current implementation of the ModPascal language environment. It comprises

- an editor for ModPascal object editing,
- a precompiler for translating the ModPascal objects to Pascal programs,
- a Pascal compiler for generation of executable code, and
- a testing device that allows execution of module operations in specific module environments.

Besides, there are a number of information commands available that e.g. list existing or accessable objects, or print them on screen, or compute interfaces for given objects.

A typical MPPS session starts with editing an object. Therefore the user has to supply an object name. If it already exists the system makes sure by request to the user, that he is willing to overwrite it possibly. If the object is new, an object type has to be supplied out of the set {MOD, ENR, INST, PROG} (standard objects are treated as modules). Then the editor is entered, and the user can type in his definition. When leaving the editor the object is created in the data base and the ModPascal code is associated to it.

Now the precompiler is invoked on the edited object. It checks for syntactic correctness of the object and performs all semantic checks that are necessary for the specific ModPascal portions of the object. Result of the invocation of the precompiler is an equivalent Pascal program.

If nothing has to be corrected, the precompiled object can be compiled by a standard Pascal compiler getting either error messages or executable code that can be used directly in a specific testing environment or is stored in the data base. If the object is involved in compilation tasks of other objects, the generated code will be used there.

5.3. Precompiling

ModPascal source code is not compiled to executable code. Instead, programs are first precompiled to Standard Pascal code, and then transformed to executable code by a Standard Pascal Compiler. The reasons for this proceeding are of practical nature: the implementation of a precompiler based on an existing compiler and runtime system ([SIEM 83]) takes less time in general than the implementation of a complete compiler. But at long term, a ModPascal compiler and runtime system has to be provided.

The most important precondition for the feasibilitv of the precompiling step is the expressibility of pure ModPascal features in Standard Pascal. Additionally, Eeery solution of this task has to guarantee that the semantics of the involved constructs are preserved.

The precompiler employed by the MPPS meets this requirements. It transforms, for example, module type definitions into a sequence of type and function definitions, or operation calls in dot notation into prefix notation. The scope of actions of the precompiler includes:

- checks of static semantics of ModPascal as described in this report and elementary Pascal static semantics
- transforming the ModPascal source code into Pascal code
- installing objects occurring in the ModPascal source code in the data base

The equivalence of source and target code then is assured by the underlaid semantics of ModPascal and Pascal, and the precompiler is designed to guarantee the equivalence. The details of the precompiling process are described in [Eck 84] (the system), [RL 85] (the transformations) and [Olt 84a] (semantical correctness).

6. Summary

The main goals the ModPascal development aimed at were defined by the objectives of the ISDV-System project. The language of the concrete level should provide structures and concepts that allow the verification of a refinement step from the intermediate to the concrete level. Beside, it should include the expressive power of an existing and recognized procedural programming language such that the acceptance of the whole system were facilitated. And finally, it should be a language which justifies its existence through the originality of its concepts alone and not through the fullfillment of the requirements of its first application environment.

From this starting point the following goals have been achieved by the development of ModPascal:

a) Design of an object oriented language with expressive features for modularization, separate compilation and hierarchization, based on a widely distributed programming language.
b) Convenient parameterization of object hierarchies by instantiation objects.
c) Provision of an elaborated environment that heavily supports the object orientedness of the language (e.g. data base for all object types).
d) The main language features can be easily connected to structures of abstract data type theory.

Especially the last point provides a promising basis for the unsolved problem in current software development systems of how to verify a refinement step that transforms an object of an abstract (applicative) level into an object of concrete (procedural) environment. Verification in this context means a mathematical proof of the validity of properties on both levels (see [Olt 84a],[Olt 84b]).

Looking at existing languages with object oriented structures, a common occurrence of a)-d) cannot be found. Often important features as incarnability of modules (ADA [ADA 80], Modula-2 [WIR 83]), object based hierarchization (Modula-2, CLU [LIS 77], EUCLID [Lam 77]) or protection of interface operation definitions (SIMULA [SIM 67]) are simply missing, and the necessary parameterization of types can only be found in the stiff form of the 'generic' construct of ADA. Additionally the underlaid semantics - if explicitly defined - do not employ special structures that reflect the object orientedness of the languages, and by this it will be difficult to incorporate them in contexts that stress verification concerns.

ModPascal has proven its adequacy for hierarchical modularized verifiable software design in case studies including practical applications as personal data management systems or accounting systems [Olt 84d].

Acknowledgement

I would like to thank my colleagues from the SEKI-Project at Kaiserslautern for stimulating discussions. This research was supported by the Federal Ministry of Research and Technology under contract IT 8302363.

7. References

[ADA 80] The Programming Language ADA. Proposed standard document, US DoD. Springer, LNCS 106, 1981.

[ADJ 78] Goguen, J.A., Thatcher, J.W., Wagner, E.G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types, in: Current Trends in Programming Methodology, Vol.4, Data Structuring (ed. R. Yeh), Prentice-Hall, 1978, pp. 80-144.

[BGGORV 83] Beierle, C., Gerlach, M., Goebel, R., Olthoff, W., Raulefs, P., Voss, A.: Integrated Program Development and Verification: In: H. L. Hausen (ed.): Symposium on Software Validation, North-Holland Publ. Co., Amsterdam 1983

[BV 83] Beierle, C., Voss, A.: Canonical Term Functors and Parameterization-by-use for the Specification of Abstract Data Types. University of Kaiserslautern, Memo SEKI-83-07, 1983.

[Eck 84] Eckl, G.: A Precompiler for ModPascal. Master Thesis (in German), University of Kaiserslautern, 1984.

[EKP 78] Ehrig, H., Kreowski, H. J., Padawitz, P.: Stepwise Specification and Implementation of Abstract Data Types. Proceedings 5th ICALP, Springer LNCS, 62(1978), 205-226.

[GHM 78] Guttay, I. V., Horowitz, E., Musser, D. R.: Abstract Data Types and Software Validation. CACM 21(1978), 1048-1064.

[ISO 7185] International Organization for Standardization: Programming Languages - Pascal. ISODIS 7185, 1982-08-12.

[Lam 77] Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., Popek, G. L.: Report on the Programming Language EUCLID. SIGPLAN, Vol. 12(2), Feb. 1977

[LIS 77] Liskov, B., Snyder, A., Atkinson, R., Schaffert, G.: Abstraction Mechanisms in CLU. CACM 20, 8(77), 564-577.

[Olt 84a] Olthoff, W.: Semantics of ModPascal. University of Kaiserslautern, Memo SEKI-84-10, 1984.

[Olt 84b] Olthoff, W.: On a Connection of Applicative and Procedural Languages. Internal Report. University of Kaiserslautern, 1984

[Olt 84c] Olthoff, W.: The Realization Level. Internal Report. University of Kaiserslautern, 1984.

[Olt 84d] Olthoff, W.: Specification and Verification of a Real-World Book-Keeping Problem with SPESY: A Case Study. Internal Report. University of Kaiserslautern, 1984

[RL 85] Breiling, M., Eckl, G., Olthoff, W., Rainau, U., Schmitt, M., Weiss, P.: The RL-Handbook. Internal Report. University of Kaiserslautern, 1984.

[SIEM 83] Pascal BS2000. User Guide (in German) SIEMENS AG, Muenchen, 1983.

[Sil 80] Silverberg, B. A.: An Overview of the SRI Hierarchical Development Methodology. In: H. Huenke (Editor): Software Engineering Environments. North-Holland, 1981

[SIM 67] Dahl O.J., Nygard, N.: SIMULA Begin. Norwegian Computing Centre, Oslo, 1967.

[SPESY 85] Schoelles, V.: The Specification System SPESY. University of Kaiserslautern. (in preperation).

[WIR 83] Wirth, N.: Programming in Modula-2. Springer 1983.