SEKI-PROJEKT

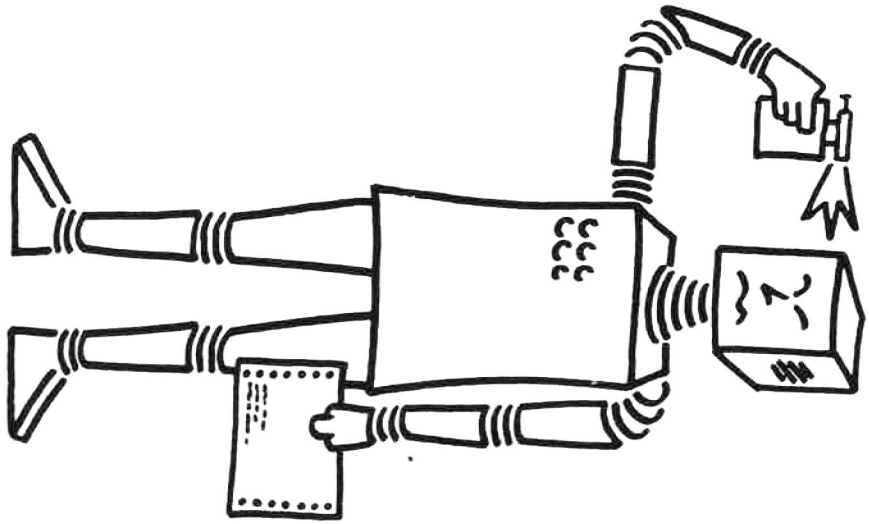
SEKI MEMO

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

On A Connection Between
Procedural and
Applicative
Languages

Walter Olthoff

Memo SEKI-85-05



On a Connection between Procedural and Applicative Languages

Walter Olthoff

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

Abstract

This paper reports on the connection between procedural and applicative languages. It presents features, notions and methods derived from abstract data type theory that in our judgement are helpful and necessary for multi-level software engineering environments in general, and especially for the treatment of verification issues there. Reference is made to an existing software engineering system and exemplary languages of it. A denotational semantics based on algebraic structures is introduced and employed. Since object-orientedness is looked at as one of the most important properties of such environments the notion of correctness is applied to objects and object relations. Finally a realistic semi-automatic method for the check of correctness criteria is given, accompanied by remarks on our existing implementation.

Contents

0. Overview

The connection of applicative and procedural languages is one of the major issues of a software engineering environment (SEE) that offers more than one language to a user. Some advanced systems include a requirement description language, a formal specification language and an imperative programming language. We refer to the entirety of a language and its support environment as Level of a SEE. Especially in the light of abstract data type (ADT) theory SEEs including an algebraic specification level and a conventional programming language level have become prominent: software development starts at abstract specifications that only consider the 'pure' algorithm or the 'pure' information of a task, and then gradually comes down to concrete programs that finally run on computers. In this scenario different languages for the description of the same thing are used; often these are applicative and procedural languages. In this paper questions as consistency of procedural programs with applicative programs and correctness are tried to be answered.

The first section introduces software engineering environments in general, and especially the Integrated Software Development and Verification (ISDV) System ([BGGORV 83], [BOV 85]). It also contains a comparison and rating of conventional Hoare-style verification vs. algebraic verification in the scenario of a SEE.

Section 2 makes the up to then informal notions of applicativeness and procedurality precise and presents representatives of these categories. The third section then provides exact definitions of syntax and semantics. Examples of language constructs are bundled in an appendix.

The main section 4 introduces so-called rep-objects and algebraic foundations that enable the definition of a correctness criterion. Also other approaches to correctness are reviewed.

In the final section 5 a semi-mechanic method for performing the correctness check is presented, and its implementation aspects are discussed.

We close with a summary and outlook to necessary extensions.

1. Introduction

1.1. Software Engineering Environments

Software engineering may be seen as the overall activity of solving problems by adequate computer programs. Nowadays an essential part of this activity is performed in specific SEEs, that provide tools and methods suited for different application areas such that a lot of cumbersome and efficiency-

reducing tasks do not take place. SEEs in our understanding are used for programming-in-the-large, i.e. the development of voluminous software products.

One important subclass of SEEs follows the 'stepwise-refinement' paradigm in program development: primarily, a problem is stated on a totally informal level, using colloquial speech, understandable both for the client and the computer scientist (we assume a scenario of this kind in the sequel). From this starting point, semiformal requirement definitions are derived that represent a helpful intermediate language with first formalizations. Then formal specifications with exact semantics are introduced; they abstract from all implementation details and describe the problem by behaviour information about operations and hierarchical relations between subproblems. The stepwise-refinement method then substitutes 'abstract' specifications by 'more concrete' ones, i.e. design decisions as data types, algorithm definitions or I/O actions are added.
This procedure goes through several iterations until a problem specification is reached that is executable on existing computers (= a conventional program).

Some SEEs supporting this technique also deal with an issue that is most important for every development of sensitive software: since each refinement step is user-defined, how can it be guaranteed that the refining structure satisfies the conditions imposed on the original structure? And in the consequence: how can it be guaranteed, that the final program does what the first formal specification of the problem demands? Only few SEEs provide assistance in the development of verifiable software in this sense (e.g. [Sil 81], [BGGORV 83]), although lack of reliable correctness concepts makes SEEs inapplicable to many practical situations. One reason for this deficiency is that verification of complex programs is impossible if no mechanical support for the check of conditions is offered; number, size and character of proof tasks generated by classical methods do not allow the manual verification of even small-sized programs in acceptable terms. But powerful automatic proof systems capable for real-word applications are difficult to design and currently available only in prototypes although many theoretical issues have been understood successfully.

In this paper we focus the attention on SEEs for development of verifiable software that follow the stepwise refinement paradigm, which in our view represents the most relevant subclass of SEEs. We assume different levels of formal problem specification, and user-defined refinement steps that transform structures of one level into structures of another.
Additionally we assume levels in which different kinds of languages are employed: an applicative language for the 'abstract' portion of the software development, and an imperative language for the final outcome of the process. This separation is justified by the fact that aspects as e.g. abstraction, representation-independence, efficiency, availibility

etc. are not satisfactory solved by a single (specification or programming) language; emphasis of one aspect comes with negligence of another. But in our view SEEs should cover them all at some point of program development and profit from each of them in order to generate verified, efficient software for existing hardware.

It is obvious that the proof of correctness criteria becomes more complicated if transitions from the abstract to the concrete level are considered. Now not only substantial changes have to be examined but also issues caused by the involvation of different formal systems. For example, a pure applicative description of something does not care for a state of a computation; if the description is expressed by imperative constructs (as assignments), it inevitably has to! - We will dedicate sec. 2 for a clarification of notions and a comprehensive discussion of issues by exemplarily presenting a concrete SEE in detail.

1.2. Hoare-Style Verification

The notion of correctness of a program is closely connected to concepts and methods developed by Floyd (inductive assertion method, e.g. [Flo 67]) and Hoare (axiomatic semantics of programming languages, e.g. [Hoa 69]). The combination of their ideas is known as Hoare-style verification. (See also, among others, [Bak 80], [Roe 76], [Dij 74] for elaborations and extensions of the approach). In their understanding the proof of the correctness of a program is equivalent to the proof of predicate calculus formulae which are generated semi-automaticaly from a program augmented by 'assertions'.

Hoare-style verification is based on the following ingredients:

- There is an imperative programming language PL (as ALGOL, PASCAL etc.) for which the notion of state as binding of variables to values is declared.
- There is an assertion language AssL which in essential is an extension of standard predicate calculus by programming language specific constructs and functions. Variables occurring in terms of AssL are PL variables, i.e. they range over domains induced by predefined types (as integer, boolean) or types generated by type constructors (as array, record etc.). The variables are interpreted identically independent of their occurrence in terms of AssL or PL.
- Elements of AssL can be elaborated in some state s by taking values s(x) for every variable x of E and applying the operation definitions. The result is a boolean value. An assertion language construct C together with a state s is called assertion.
  Notation: Cs.
- Hoare semantics for programming languages are based on formal systems. A formal system is a tuple (FORM, RULE) where FORM denotes a Herbrand universe over a given set of

symbols (the formulae), and RULE a set of rules $F_1 \vdash F_2$ that allow to syntactically decuce formula F_2 from formula F_1; $\vdash F$ is called axiom.
For the semantics of PL a set of proof rules of the form

$$P_1, \ldots, P_n \vdash C_1, \ldots, C_m$$

is given where the premises P_i and conclusions C_j are correctness expressions of the form '{P} constr {Q}', P,Q ∈ AssL, constr ∈ PL. Then a proof theory PT(PL)=(Corr,R) is defined as a formal system with correctness expressions Corr as formulae and proof rules R as rules.

An interpretation I: Corr ⟶ State ⟶ {true, false} assigns a meaning to correctness expressions:

$$I(\{P\}\ constr\ \{Q\})s := \begin{cases} true & \text{if (Ps ==> Qs', constr transforms s into s')} \\ false & \text{otherwise} \end{cases}$$

s,'reasonong)'='reasoning and assumptions on

PT(PL) is assumed to be sound and complete with respect to I. Note that AssL is also connected to a sound and complete proof theory PT(AssL) where the rules are standard logic deduction rules.

- The effect of every construct of PL is described by rules of R. Therefore in the correctness expressions of the rules technical constructions such as substitutions of variables, introduction or splitting of assertions special notions or pure implications (P==>Q) are used to formalize the intended meaning. For example, the rules for assignment frequently are

  $\vdash \{P\langle x \leftarrow e\rangle\}\ x := e\ \{P\}$ or
  $\vdash \{P\}\ x:=e\ \{P\langle x \leftarrow e\rangle\}$

  (dependant of backward or forward reasoning) where $P\langle x \leftarrow e\rangle$ denotes the substitution of all free occurrences of x by e in P. (In fact, this rule is an axiom in PT(PL)). Unfortunately some features are only covered either with restricted applicable or unusable complex rules (e.g. side-effects / global variables, procedure and function declarations and calls, iterative structures). But despite of these limitations there is a proof technique that employs the rules of R in backtracking, subgoaling and unification steps in order to generate from a given '{P} constr {Q}' ∈ Corr a (set of) AssL formulae (see [STA 79] for an exemplary implementation).

With (at least) these ingredients Hoare-style verification works as follows:

1) Suppose there is a program prog that is intended to solve some problem. The goal is to formally prove that this is in fact the case. The first step consists in a formalization of the intention: the programmer has to to state P,Q ∈ AssL such that P holds before execution of prog, and Q thereafter. Note that from this point only P and Q are the relevant benchmarks; there is no 'verification' whether P and Q meet the intention! (This problem is often brought up by critics of the approach; but the transition from (immaterial) intention to (material) formalization will never

be verifiable in the usual sense, independent from the specific approach. So this objection (sometimes called 'the immanent bias') is not constructive, and we judge it senseless).

2) P and Q are also called input and output assertion resp. and usually they do not describe properties of operations of prog but values of variables before and after execution. Beside the input and output assertion, other assertions have to be defined by the programmer: for procedure and function declaration bodies, entry and exit assertions have to be supplied that describe - similarily to the input and output assertion - the behaviour of the operations body. Both assertions are used in the rules for procedure and function calls. Also, for each iterative structure (while, repeat), invariant assertions must be stated. Invariants are true whenever control flow passes them. They represent inner properties of loops, that are exploited by the programmer in his algorithm (in prog). Finally, (arbitrary) free assertions may be stated, if it is viewed at as a necessity to get a correctness decision, or to better document prog, or other reasons.

3) All assertions except the latter (input, output, entry, exit, invariant) are inserted in the prog code at predescribed positions; free assertions may be inserted between arbitrary statements. Thereafter, instead of prog an 'annoted' program prog' with inserted assertions is considered.
Note that the correctness of the program is checked not only against the input/output assertion but also against all entry, exit, invariant and free assertions. They all together constitute the formalization of the intention, and with them numerous possibilities of introduction of immanent bias are offered. Additionally, invariants will seldom allow proofs of correctness if they are not 'strong' enough i.e. if their extensions cover too few cases.

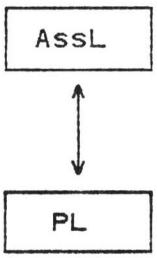
4) With P(input), Q(output) $\in$ AssL, and prog' an annoted program derived from prog $\in$ PL we now consider the correctness expression
cexp := {P} prog' {Q}.
Prog' will be called <u>correct</u> w.r.t. P and Q, if I(cexp)s = true holds for all states s. This is equivalent to: cexp is derivable in PT(PL) such that it is sufficient to construct a derivation of cexp from R.
Rules like the assignment axiom correspond directly to (sets of) assertion language formulae: P ==> P <x $\leftarrow$ e>, and this holds for every rule of R. Therefore a deduction in PT(PL) is equivalent to a set of assertion language formulae, the so-called verification conditions (VCs). If all VCs can be shown valid (i.e. deducable in PT(AssL)), then Qs' holds; or in other words: I(cexp)s is valid and prog' is correct.

5) A mechanical theorem prover is employed for the proof of

the VCs. Additional lemmata can be inserted in a 'knowledge base' which already contains axioms and rules about AssL.
If the proof attempt fails, then either
- entry/exit assertions were inadequate/wrong, or
- invariants were too weak or wrong, or
- free assertions were inadequate/wrong, or
- (the most interesting case) prog does not what is specified in P and Q.

The detection of the last alternative represents the major goal of program verification efforts: a mathematically precise proposition about an erroneous program, that possibly provides hints for malfunctioning code lines. Unfortunately, it is the exceptional case that exactly the last alternative is applicable; more often all alternatives contribute to the unsuccessfull proof attempt, and the programmer is encouraged to change program or assertions or both.

Pictorially, Hoare-style verification encloses two levels:



specification (expressions)

correctness

solution

Here, the classification of SEEs of the introduction above can already be motivated: AssL is an extension of predicate calculus, a highly applicative specification language, and PL is by definition imperative. So Hoare-style verification is a possible (rudimentary) incarnation of SEEs we consider here.

What are the disadvantages of this approach that hinder its practical application in SEEs?

- Firstly, due to historical reasons, Hoare-style verification is dedicated to 'reverse software development':
  One starts with an already written program in which algorithms are designed along concrete data structures, and annotes it with its intention. Nowadays people go the other way around, for various reasons (that are skipped here).
- Secondly, the whole theory knows only one refinement step from AssL to PL. This is an unfeasible way for every larger software development, since the inherent complexity and hierarchical structure of a problem has to be covered and equalized, often an impossible requirement.
- Third: This approach is not object-oriented - in contrast to the widely accepted benefits of this programming style. Constructs of AssL and PL are formulae and programs(fragments), and the approach reflects no results of abstract data type theory (e.g. the grouping of data and operation in one structure and algebras as basic semantical concept). Currently, proof rules for objects and object operations are topics of research, but a successful answer is still open.

- **Fourth:** No concepts for parameterized structures have been developed. Together with a missing notion of implementation two important and necessary features for software development in the large cannot be used.

From all this it comes out that classical Hoare-style verification is bad suited for SEEs for development of reliable software.

1.3. Algebraic Verification

Employment of algebras and algebraic structures for verification purposes (algebraic verification) represents an alternative to classical approaches. This way is heavily based on results of abstract data type (ADT) theory as developed for example in [ADJ 78], [EKP 78], [ADJ 79], among many others. In ADT theory the only occurring semantical structures are algebras and algebra morphisms (or more generally: functors), and concepts like implementation, parameterization or verification are based upon them (e.g. correctness of an implementation is often defined by special algebra homo/isomorphisms). In many cases verification is equivalent to correctness criteria expressed in algebraic terms, and a verified structure is one that satisfies these criteria.

Algebraic verification then is used in two contexts:
- verification of relations between structures of a specific level of a SEE
- verification of relations between structures of two different levels of a SEE.

As pointed out above, we concentrate ourselves in this paper on the second application, and more precise: verification of relations between a structure of an applicative level of a SEE and a structure of an imperative level of a SEE.
A characterization of algebraic verification in this context involves (at least) the following features:

- There is an imperative programming language PL for which the notion of state as binding of variables to values is declared.
  Also the language provides constructs for object-oriented programming, features for hierarchization of objects, and concepts for parameterized objects.

- There is an algebraic specification language SL whose basic structures are algebraic specifications. This guarantees the possibility of object-oriented programming.
  Additionally, the language allows hierarchical specifications and provides concepts for combination, parameterization, and implementation.

- The semantics of PL is based on the idea of state transformations caused by PL-constructs. To describe it a traditional denotational semantics DS(PL) is assumed. From this

an algebraic semantics AS(PL) of PL is derived by:
- association of every pre- or user-defined type definition with an appropriate algebra
- association of every object definition with an algebra
- association of state transitions caused by operation calls with algebra operation calls
- association of concepts as parameterization or implementation with special algebra morphisms and algebras.

The standard semantics has to be enriched by appropriate domains to achieve this modification. AS(PL) is assumed to describe also issues as side-effects or pointer types, scoping, typing of expressions, and so on.

- For SL semantics, ADT theory provides several choices: initial, terminal, loose and variants thereof, and each approach is praised by its apologists. Since SL is employed in a SEE we require the following mixture: To postpone representation decisions as far as possible (i.e. to the final refinement steps) the SL semantics has to be as abstract as possible. This can be achieved, if specifications are supplied with a loose semantics: then the meaning of such an object is the set of all models (= algebras, that satisfy certain conditions) of it. Since in general algebras should serve as semantics of objects, a loose semantics like this possesses a high degree of abstraction and flexibility since no possible model is excluded. Also, specifications may be compared by looking at their set of models, and concepts as refinement, implementation or parameterization can be described by mappings between such sets.
  The stepwise-refinement method includes that final structures are reached on the applicative level which are no longer subject to refinement. For specifications with this property it is unwanted to have a set of algebras as semantics but a single unique algebra. The SL semantics we assume provides this property for certain specification objects.

- Since PL and SL are semantically based on algebraic structures, a refinement situation in a SEE involving a SL specification and a PL object is just a relation between (sets of) algebras. Therefore a correctness criterion attached to this relation should also be expressed in algebraic terms. For software engineering purposes, one is interested that refining and refined structures behave equivalently i.e. there exist homomorphisms or isomorphisms between the associated algebras. Even if homomorphy seems to be the weaker requirement, it is sufficient as correctness criteria for the SL-PL refinement (see also the remark after definition 4.4.-2).

It should be clear that this is an extremely brief and superficial presentation of prerequisites for algebraic verification, and that the attention is focussed only on ideas that are helpful in the explanation of level-transgressing refinements in a SEE. Necessary precise definitions are supplied in the subsequent sections.

With (at least) these ingredients algebraic verification of refinement relations between structures of two different levels of a SEE works as follows (see also section 4.):

1) Suppose there is a specification S that describes a problem solution in SL. S is viewed as final, such that only a refinement into a PL structure has to be done. Note that S is the formalization of the intended problem solution. From now on only S is the relevant benchmark; there is no verification whether S meets the intention (see the remark in paragraph 1) of section 1.2.).
Let O denote an object of PL, that is intended to represent the refinement. O differs from S in that concrete data structures are introduced and algorithms are defined operating on these data structures and all efficiency-increasing features of PL are exploited.

2) To be able to connect S and O it is necessary to get more information: which data in O refines operations of S? The programmer has to supply the intended associations that have to respect some basic requirements (e.g. preservation of operation functionalities).

3) Referring to data associations additional information is needed. S as well as O are semantically described by unique algebras A(S) and A(O), and since it is in general impossible to automatically construct homomorphisms between arbitrary algebras, the programmer has to make a suggestion. From his knowledge of the intention behind the refinement step and the details of his objects S and O he is able to specify how concrete data realizes abstract data, that is to define a mapping M: A(O) ——> A(S) that applied to a concrete carrier element of A(O) yields an abstract carrier element of A(S).

4) From the information gathered in steps 2) and 3) it is possible to formulate a set of (homomorphy) equations. The general scheme is:
M(O_op(arguments)) = S_op(M(arguments))
where O_op and S_op are associated by step 2). If all equations can be shown valid, then M is a homomorphism, and one gets the desired result about S and O.
It should be noted that the verification is not performed solely with respect to S but also against the associations of step 2) and M. All items together constitute the intention behind the refinement, and each of them offers a possibility of introduction of immanent bias.

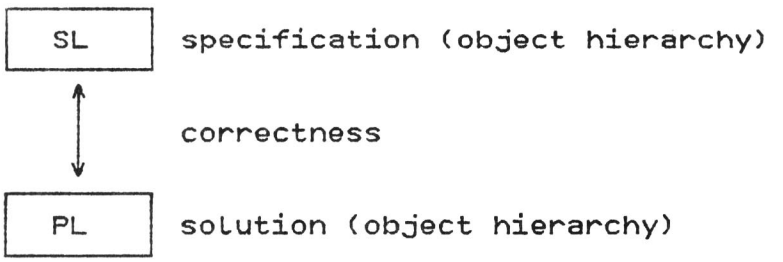
5) To show a set of equations valid in some theory requires some mathematical apparatus. Also constraints have to be considered that come from the hierarchical structure lying on object sets (e.g. only those proofs will succeed if all substructures of involved objects are 'correct'). Depending on the characteristics of the applied proof system (equality prover, induction prover, etc.) the preconditions and techniques vary. Therefore we skip details and refer to

section 5. where a special constellation and solution are presented.

If the proof attempt fails, then either
- the associations of step 2) were inadequate/wrong, or
- S was inadequately or wrongly defined, or
- O does not what is specified in S.

If one is sure of the last alternative, algebraic verification has paid off: one has a mathematically precise proposition for an inadequate refinement in which required properties get lost. Unfortunately, in general it is unclear which alternative causes the failure. Moreover, all alternative possibilities contribute to unsuccessful proof attempts.

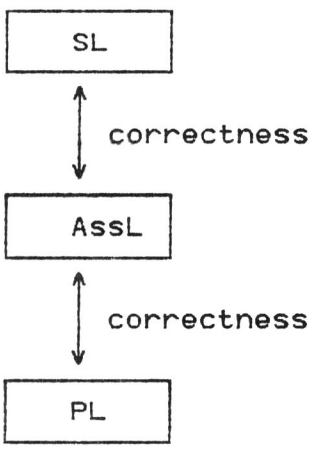
Algebraic verification (as described above) can be summarized and visualized as follows:



This configuration strongly ressembles to SEE pictograms, and in fact, this approach can be directly applied in the SEE subclass described above.

Compared to Hoare-style verification, all disadvantages listed in sec. 1.2. are missing: 'forward software development', multi-step refinement, object-orientedness and parameterization/implementation concepts are essential features and central issues in the algebraic approach. Many useful concepts of ADT-theory are made accessable to non-experts that use SEEs build on this foundations.

At first glance, it would be nice to unify both approaches: extend AssL of sec. 1.2. to SL and use the existing apparatus for Hoare-style verification:



But as the picture shows, this comes with a new correctness problem: SL constructs have to be refined into AssL constructs (if the Hoare-style verification part is left unchanged), and this is a degree of additional complexity that is highly unwanted. Moreover, one can doubt if an embedding of an object-oriented algebraic specification language in a formula-oriented assertion language is possible at all.

On the other hand, if one wants to modify AssL and PL to overcome some of the problems basic research on Hoare logics has to be performed: proof rules have to be developed for object declarations, for object incarnations, for declarations and calls of operations of objects, for parameterization of objects, etc.; the assertion language has to cope with these extensions; associated proof theories have to be shown sound and complete, and so on. But the main point is that even after successful completion of the above agenda, Hoare-style verification will still be inadequate. It is still backward software development, and the basic idea is still to model the state change caused by PL constructs and then show some assertions and implications valid. How the single proof tasks correspond to some property of the specified problem gets lost because many formulae are introduced for technical reasons (e.g. decomposition rules, iteration rules). This is far less than offered by the algebraic approach where each proof task can be logically assigned to some subproblem. Moreover the proof tasks are of deeper mathematical quality than predicate calculus formulae and therefore allow more powerful propositions about the generated software.

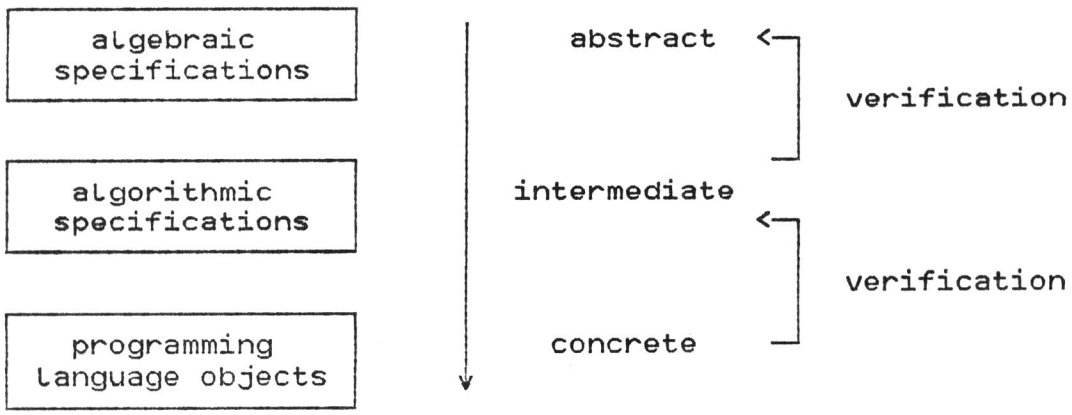
In the following we employ the algebraic verfication approach.

1.4. The ISDV-System

This section gives an overview on the Integrated Software Development and Verification - (ISDV-) System ([BGGORV 83]). The ISDV-System is a SEE that meets our confinements; most concepts presented here were originally designed and implemented for that system. 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 1.4.-1 a rough overview of the various levels is given together with a also rough classification, and the verification tasks are located.



1.4.-1 Fig.: 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]) but realizes the 'loose-semantics' approach ([BG 77], [HKR 80]). 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.

The abstract specification level can be characterized by three sublevels with different degrees of abstraction:

- the axiomatic level (AX)
- the algorithmic level (ALG)
- the intermediate level (AX/ALG)

AX comprises what is known as '(axiomatic) algebraic specifications': objects which are defined by indication of a signature (a set of sort names and a set of operation names with arity) and a set of PC-1 formulae ('axioms') built from the symbols of the signature. Every problem specified on AX is a set of axiomatic specifications, possibly hierarchically connected, where the semantics of operations are described by the axioms of the specifications. There is, by definition, no information about control flow or sequentialization in AX specifications; a (more or less) explicit definition of an operation can only be received by applying the semantics generation mechanism that is associated to one's (axiomatic) specification language (initial/terminal algebra semantics, model-theoretic semantics, rewrite rule semantics or whatever). Often, the result of such an application is hard to compute

and hard to use.

On ALG, the demand of representation-independence is slightly relaxed: operation definitions are still based on terms built from symbols of the signature (as in axioms of AX objects), but they are stated in an algorithmic manner employing controlflow constructs as 'if-then-else', 'case' branchings or recursion. The algorithmic definition replaces the axiomatic definition of an operation, and in general it gives rise to a unique semantics that can be generated by least fixpoints of functionals. With an appropriate environment (interpreter), specifications of ALG become directly executable, and testing of 'abstract programs' then is the task of evaluating terms, which is feasible by applying the operation definitions associated to the operation names occuring in the term.

The intermediate level AX/ALG consists of those objects that involve AX- as well as ALG-subobjects.

On all levels it is possible to refine or implement objects into other objects; the main problem is to ensure the preservation of semantical properties during the establishment of a refinement or implementation relation.
For example, AX and ALG objects are interwoven with each other in two aspects:

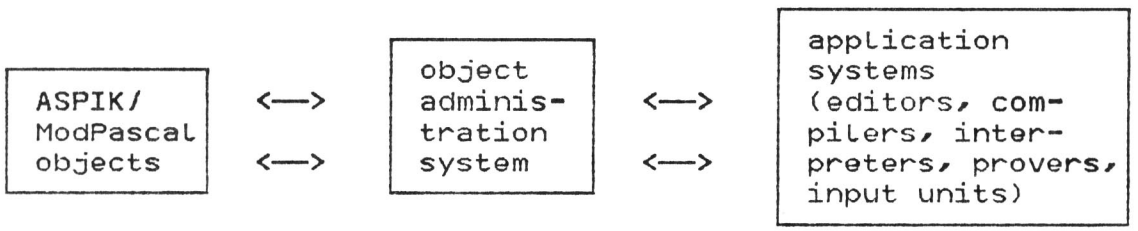
- Every axiomatically specified object is transformed during the refinement process into an algorithmically specified object. This technique borders the increase of complexity of refinement steps by only allowing modification of operation (and not simultaneously modification of data).
- There is a notion of correctness of a refinement step from AX to ALG: The algorithmic definition of an operation - up to now in no way related to the axiomatic definition of the identically named operation in AX - is required to fullfill the axioms of the associated AX object. If this can be guaranteed, both definitions describe (at least) overlapping functions, and the refinement is semantics-preserving.

The concrete level has to provide a language that allow both enough expressiveness and allow efficient programming on von-Neumann machines. Expressiveness means that the problem solution of the abstract level - as it is visible in the structure and number of specification objects in AX and ALG - has not be reinvented, but can be carried over. Therefore it is necessary to have specification-like constructs.
Efficiency considerations are not much emphasized when looking at theoretical issues of software development environments. But for real-world applications and acceptance it is indispensable to be able to 'tune' program code e.g. by replacing recursion through iterations in order to optimal use of hardware ressources. Therefore, the imperative language of the concrete level should include major subsets of languages like ADA or Pascal.

For completeness it should also be mentioned that the ISDV-

system software development scenario also includes tools that allow to enter abstract or concrete programs, to establish refinements and attach semantical properties to them, to generate prooftasks that are submitted to connected mechanical proof systems, and above all a sophisticated object administration system that does the 'dirty work' of data base management and of semantical property maintainance for objects of all levels. The data base contains pre- and user-defined objects and it helps to avoid 'development-from-scratch' because it is allowed and encouraged to re-use already defined objects in different applications. Also library puroses are supported by it.

Logically, we have the following system structure of the ISDV-System:



1.4.-2. Fig.: ISDV-System structure

One of the proof tasks mentioned above is described in this paper: the algebraic verification of refinements of algorithmic ASPIK objects into ModPascal modules. Further details about other proof tasks and the ISDV-System can be found in [BV 85], [Sch 85] or [RL 85].

2. Applicative and Procedural Languages

The previous section introduced the scenario, in which we are going to develop our approach. Now we make more precise our notion of applicative and procedural languages (sec. 2.1.). Then the main problems of a connection of both formalisms are presented (sec. 2.2.) and finally two representatives of the language families are briefly introduced (sec.s 2.3. and 2.4.); exact definitions are given in sec. 3.

2.1. Classification

In the following we try to partition the set of existing programming languages (PL's). This takes its justification from the fact, that almost every application area of computer science has developed a preference for a specific set of PL's: economy and buisiness located tasks are programmed in e.g. COBOL, numerical applications are preferably written in e.g. FORTRAN, ALGOL, PASCAL, process automation is supported by e.g. PEARL, concurrent programming is performed e.g. in ADA, or artificial intelligence problem solutions are heavily based on LISP and PROLOG. The benefits of these associations of application areas and languages will not be discussed here since

the classification we are aiming at is more general.

We will distinguish three categories of PL's where the description of each category is given below:

- applicative PL's
- procedural PL's
- other PL's

The term applicative PL refers to a language with at least the following properties:

a1) There is no concept of global variables.

a2) There are no "assignment-constructs", and the semantics is not based on states and state transitions.

a3) The control structure of a program is definable only by means of conditional branching and recursion.

a4) The concept of action (i.e. how things are sequentialized in time) includes only function composition and function application.

a5) The concept of data consists of a set of so-called elementary objects and associated functions (see remark d) below). Note that strong typing is not induced herewith.

The term procedural PL refers to a language with at least the following properties:

p1) There is a concept of global variables (of a program or operation).

p2) There is an "assignment-construct" which changes effectively the value of an assignable object, and the language semantics is based on states and state transitions.

p3) The control structure of a program is definable by means of conditional branching, iteration, jumps, and recursion.

p4) The concept of action is sequences of statements (state changing actions).

p5) The concept of data consists of predefined types, predefined type generators, functions, and procedures.

PL's that do not fall into one of the above categories are referred to as other PL's.

This (and every) classification cannot be clean, exhaustive, or unique. Already the pretention of exactness and completeness of the listed properties may raise opposition, and we are conscious about this. On the other hand, every other proposal will have to deal with the above criteria more or less explicitly, possibly adding or removing specific points, or putting emphasis on different requirements. For example, approaches to the semantics of applicative languages may employ lambda calculus, data flow models or reduction processes, but will always be 'non-state based'. Objections of this kind will not question the necessity of any classification for our context, and therefore we will use the introduced terms as intended without glancing over their deficiencies.

To make precise our understanding of procedural and applicative PL's, we add some remarks.

a) Existing practically used languages are seldom purely procedural or purely applicative. For example, it is difficult to characterize INTERLISP as applicative because there are asignments, loops and goto's among the language constructs. These features were introduced into the applicative PL (PURE) LISP to overcome alleged shortcomings compared to FORTRAN and to popularize applicative programming.
   Procedural PL's on the other hand frequently include features as functions or recursion (e.g. PASCAL) so that the adjustment to some PL class becomes questionable. But despite this classification problem for existing PL's we will maintain the categories because they highlight theoretical problems that occur in multi-level software development environments employing different language types.
b) On todays machines, programs written in (more or less) applicative PL's are not supported by the hardware architecture. Conventional von-Neumann computers are designed for tasks described in procedural PL's, and when using other kinds of description languages one is finally forced to compile one's description into the machine language. This also has caused the proliferation of state manipulating constructs in applicative PL's.
   Applicative PL's need appropriate machine support to exploit their theoretical properties and convenience. As long as appropriate and powerful hardware is not developed or available, applicative PL's will increase their involvement of procedural PL concepts to remain competetive.
c) We classify some existing languages as follows:
   applicative: PURE-LISP, INTERLISP, PROLOG, FPL, APL;
   procedural: ALGOL, FORTRAN, ADA, PASCAL, MODULA-2.
d) The term 'functional PL' often refers to languages that are applicative PL in our sense, with the modified property a5) that allows also functions as data objects (all LISP dialects are 'functional').
e) The concept of action for applicative PL's is essentially the building of expressions and the evaluation of them. In procedural PL's also expressions may occur but they are substructures of statements and are only usable in this context (i.e. vehicle for the formulation of a state transition).

2.2. The Connection Problem

Procedural and applicative PL's have advantages in specific problem areas, as indicated above. In general one need not consider any interactions or relations between them.

In the environment of software development systems employing various languages and/or languages of different kinds the separate view is no longer possible. Objects (pieces of programming language code) of some stage of the development are linked to other objects of other stages, and one is inter-

ested to state properties of such links (e.g. refinement links, implementation links, description links). If languages are changed within a link between a source object and a target object, the connection problem (CP) occurs:

> How can some link property be formalized and how can the formalized property be verified?

Verification in this sense means to show the somehow defined validity of a correctness condition.
The situation becomes extremely severe if the link property states a kind of semantical preservation, i.e. the source and target object of a link are intended to be equivalent even if they are expressed in different languages. This occurs in general in software development systems that start with requirement definitions of a problem and end with executable code: everything remains critical unless the fullfilment of the requirements by the generated code is not assured.
In the sequel, the CP is examined under the additional assumption that the object descriptions are given on the one side in an applicative PL, on the other side in a procedural PL. Then, taking the specific properties of applicative and procedural PL's as defined in sec. 2.1. into account, (at least) the following problem areas may be located that aggravate a connection of the type "semantically equivalent":

CP1: Side Effect Freeness vs. Global Variables
Typically, data of applicative PL's consists of set(s) of elementary objects and functions defined on them. The former can also be viewed at as constants or no-argument functions delivering itself as value. The language allows the composition of expressions from this data such that the value of the expression is derivable only from the expression and the function definitions (this property is sometimes called 'referential transparency').
This is not true for expressions or other pieces of code in a procedural PL. Both may involve (a set of) global variables that will not have an explicit value by the actual parameters supplied to operation calls of the expression (= piece of code). Therefore the meaning is only derivable in the context where the global variable values are known.
CP1 then may be formalized as: Can expressions of an applicative PL and statements of a procedural PL be compared (in the sense of "semantically equivalent"), and if so, what are the conditions?

CP2: Functions vs. Procedures
In fact, this is a subcase of CP1 but an interesting one. If an applicative PL function and a procedural PL procedure are considered the global variable and expression/statement questions are again raised. But looking at the operation definitions, now can a statement "the function is equivalent to the procedure" or "... does the same ..." be established in a formal setting? This problem occurs everywhere in SEEs when operation definitions are optimized or reformulated on different data structures.

CP3: Object Oriented Semantics
Applicative PL's used in software development environments often come with compound syntactical structures that enclose data as well as operations and that alleviate clustering of tasks into coherent units. Sometimes these units can build hierarchies such that for a hierarchy element all depending objects are visible and their contents (data, operations) are usable in it. Languages with these poperties are often called object oriented, and there are also a number of procedural PL's that meet the above description more or less exactly (e.g. ADA, CLU). Then, the connection problem for unit definitions is on the one hand independent of the kind of language, on the other hand it is aggraveted in our software development context by CP1 und CP2; the problem may be stated as: "is a unit definition in an applicative PL 'semantically equivalent' to a unit definition in a procedural PL?" Or more practical: "is a given unit definition a semantic preserving implementation of another one?"

Depending on the point of view, more or other connection problems may be recognized. Whenever the verification aspect is not stressed, the connection problems are probably solvable with satisfactory concepts and pragmatic decisions. In the other case a formal mathematical framework has to be set up in which applicative and procedural PL's become comparable and notions as "semantically equivalent" or "correct implementation" can be introduced naturally. Without a formalism of this kind the software development verification problem ist not solvable.

2.3. The Applicative PL ASPIK

In this section we give a brief overview on our version of the applicative PL ASPIK that differs from the version used in the ISDV-System (see sec. 1.4.). The overview covers only the most relevant features of the language. This section introduces their syntax, whereas a (partial) formal semantic definition is given in section 3. A full description of ASPIK may be found in [Lic 85], [Spa 85] and [Sch 85].

The development of ASPIK was heavily influenced by abstract data type theory. Especially the notion of algebraic specifications (see e.g. [ADJ 78], [EKP 78]) had formed the morphology of the language. ASPIK distinguishes between three kinds of objects:

- specifications (axiomatic or algorithmic)
- maps (refinement or implementation)
- imps (signature or specification)

Specifications are named syntactic units that allow the definition of data - the 'sorts' of the specification - and operations; on the other hand maps are named syntactic units to associate sort and operation names in different specifications. This association is necessary for the refinement, parameterization and implementation concept of ASPIK. Imp objects are used to realize the ASPIK implementation concept. They specify certain properties of specifications that are

said to implement each other. Specifications, maps, and imps can be structured hierarchically, i.e. they include special 'use' - slots for indication of all those objects the current one is built upon. Since the use-relation inheres in a direction, a hierarchy of ASPIK objects can be visualized by an acyclic directed graph. The requirement of acyclicity excludes recursive object definitions from the language.

For the purpose of this paper it is sufficient to focus the attention on specifications since only the connection of this object category to a procedural counterpart is examined. Therefore we will only present an overview on maps and imps, but go into details of specification objects.

Specification objects consist of a header and a body. If only the header is defined, the specification is called axiomatic (or 'loose'; see [BV 85]), otherwise algorithmic.
A specification object in ASPIK is composed of a number of mandatory (m) and optional (o) clauses:

specification header (m):
: consists of maximally the specification identifier, use-, sorts-, ops-, and props-clauses. The header describes the interface of the object, i.e. the names of sorts and operations visible to the environment:

specification identifier (m):
: a unique name for the entire object; also used in some contexts for prefixing of identifiers that are introduced in the current definition (cd).

use-clause (m):
: a list of objects that are used, i.e. their sorts and operations may occur in the cd. The objects are either referenced by a specification identifier or by a specification term (spec-term; see below).

sorts-clause (o):
: a list of new sort names. The sort names can be used in the rest of the specification, e.g. to define functionalities of operations or to indicate the scope of variables in the prop-clause.

ops-clause (o):
: a list of operation functionalities of the form
op: $sort_1$ $sort_2$ ... $sort_n$ --> $sort_{n+1}$
The operations are viewed at as interface or public operations that are visible inside the cd and in all objects that use the current object. The ops-clause contains no explicit operation definition.

props-clause (o):
: The properties clause consists of axioms which are predicate calculus formulae that are intended to describe the behaviour of the operations of the ops-clause (Note: no concrete definition is given for operations, only functionalities). There are no rules how to specify the intended behaviour; often equations are used to express the operations semantics axiomatically (although the term 'axiomatic specification' ambiguously encloses every body-less specification in-

dependent from the form of the props-clause).
The props-clause content of a specification serves as a formal requirement definition for possible algorithmic operation definitions of those operations introduced in the ops-clause. Several correctness criteria for specifications are connected to the relation between properties and algorithmic operation definitions (see [BV 85] and below).

specification body (o):
consists maximally of the constructors-, auxiliaries-, define-auxiliaries-, define-carriers-, define-constructor-ops-, private-ops-, and define-ops-clause. In the specification body algorithmic definitions for newly introduced sorts and operations of the cd are given. To facilitate this, one can define auxiliary and private operations which have limited scope.

constructors (m: the sorts-clause is nonempty):
a subset of the ops-clause operation identifiers. The constructors are the generators of the Herbrand universe that is used in the definition of carriers. Each constructor contributes to the Herbrand universe that is associated with its target sort name. Herbrand universes are considered for each sort name occurring in the sorts-clause.

auxiliaries (o):
a list of operation functionalities in the same form as in the ops-clause. An auxiliary operation is intended to ease the definition of carrier sets in the define-carrier-clause. The scope is restricted to the specification body.

define-auxiliaries (m: the auxiliaries-clause is non-empty): gives the concrete definition for all auxiliary operations. Occurrences of conditionals, case- and letschemes, recursion and operation calls are allowed in auxiliary operation definitions. Visible items are all used sorts and operations, newly introduced sorts and newly introduced operations.

define-carriers (m: the sorts-clause is non-empty):
for each sort name of the sorts clause, a carrier set is defined by the extension of a so-called characteristic (carrier) predicate. The extension represents either the complete Herbrand universe, spanned by associated constructors, or a subset of it and it includes a special error element that is different from all other elements. In the subset case provisions have to be taken to guarantee the well-definedness of operations (closure property; see below).
In the definition of the characteristic predicates, an operation of the ops clause is unvisible. Used and auxiliary operations are visible. The carrier predicate definition consists of an arbitrary operation scheme (conditional-, case-, let-scheme or term) that satisfies the subterm property (occurring terms are carrier elements) and that evaluates to a boolean value:

- true: the argument term is carrier element
- false: the argument term is not carrier element

It should be emphasized that the carrier definitions are a very crucial part of an ASPIK specification since they determine the data of the abstract type behind the specification and therefore influence the consistency of the cd with the intended model as it is described in the specification header (especially the props-clause; see sections 3.2.1. and 3.4.1. for the semantics of specifications).

define-constructor-ops (m: the constructor-clause is non-empty): here, the constructors given in the constructor clause only by name are completely defined. There is limited freedom in the operation definition since the decisions of the define-carrier-clause have to be respected. The main point is: if the carrier definition has specified a term $op(a_1, ..., a_n)$ as carrier element, then an invocation of the operation op on the arguments $a_1, ..., a_n$ has to evaluate to this term (note the subtle distinction between terms and invocations of operations that are associated to the names occuring in the terms). Otherwise the value of a constructor operation invocation has to be defined such that closedness is maintained i.e. if the carrier predicate has excluded a constructor term then the associated constructor operation call has to be defined yielding an element of the carrier (in the trivial case: the error element). Constructor operation definitions may depend on characteristic predicate definitions. Visible items are the used operations, the auxiliaries and the constructors of cd, and the definition may be based on case-schemes, let-schemes, conditionals, recursions and operation calls.

private-ops (o):
introduces functionalities of operations that are intended to be used in the operation definition of public operations but should not be accessible outside the specification (hidden operations). They are similar to auxiliary operations but with different application area (definition of public operations instead of definition of carrier predicates). The functionalities may be built up from used sorts and sorts of cd.

define-ops (m: ops- or private-ops clause is non-empty):
all operations up to now only introduced by functionalities are defined. Visible items are all used operations, all sorts, operations and private operations of cd. The definition may include occurences of case- and let-schemes, conditionals, recursions and operation calls.

This brings the overview on ASPIK specification objects to its end.

2.3.-1 Example: Limited Queue specification in ASPIK

```
spec QUEUE;
use INTEGER, BOOLEAN;
sorts queue;
ops emptyqueue: --> queue;
    enter: queue integer --> queue;
    remove: queue --> queue;
    first, last: queue --> integer;
props [P1] all q: queue all i: integer
           remove(enter(q,i)) == q
      [P2] all q: queue all i: integer
           last(enter(q,i)) == i
      [P3] all q: queue all i: integer
           q == emptyqueue
            ==> first(enter(q,i)) == i
      [P4] q =/= emptyqueue
           ==> first(enter(q,i)) == first(q)
```

```
spec body
constructors emptyqueue, enter;
auxiliaries
            size: queue --> integer;
define auxiliaries
            size(q) = case q is
                      *emptyqueue : 0
                      *enter(q1,i1) : succ(size(q1))
                      esac
define carriers
            is-queue(q) = case q is
                          *emptyqueue : true
                          *enter(q1,i1) : less(size(q1),10)
                          esac
define constructors
            emptyqueue = *emptyqueue
            enter(q,i) = if less(size(q1),10)
                         then *enter(q,i)
                         else q
define ops
    remove(q) = case q is
                *emptyqueue : error.queue
                *enter(q1,i1) : case q1 is
                                *emptyqueue : emptyqueue
                                *enter(q2,i2) :
                                enter(remove(q1),i1)
                                esac
                esac
    last(q) = case q is
              *emptyqueue : error.queue
              *enter(q1,i1) : i1
              esac
    first(q) = case q is
               *emptyqueue : error.queue
               *enter(q1,i1) : case q1 is
                               *emptyqueue : i1
                               *enter(q2,i2) : first(q1)
                               esac
               esac
```

endspec

Remarks a) Empty clauses are skipped
b) INTEGER and BOOLEAN denote the specifications of the obvious objects.
c) The props-clause should also reflect the limitedness of queues; appropriate equations are disregarded to support compactness of the representation.
d) Succ denotes the successor operation of INTEGER.
e) Starred items denote carrier elements: while emptyqueue is an operation of QUEUE, *emptyqueue is an element of the carrier queue of QUEUE.
f) Less denotes the obvious operation of INTEGER.
g) Enter is defined to deliver its queue argument unchanged if the maximal size is reached. FIRST evaluates to the innermost integer argument, which in turn is removed by REMOVE.

∎

Two topics were spared in the introduction of specifications above: the subterm property and spec-terms.

The subterm property is very crucial for the definition of carriers and constructors. It says that whenever a term t consists of operation symbol op and argument terms t_1, ..., t_n, then it holds: if t is carrier element, then t_1, ..., t_n are carrier elements. This property ensures for example the well-definedness of the carrier predicate since subterm extractions do not violate the closedness of the predicate, and also recursive invocations are defined. In the introduction above the subterm property has been omitted for reasons of clarity.

Spec-terms are a very important feature of ASPIK. They involve specifications as well as maps or imps. For a precise understanding of spec-terms we give a short survey on map and imp objects in ASPIK which both are intended to establish relations between specifications. The essential notion behind maps is the notion of signature morphism. Signature morphisms are pairs of mappings between sets of sort names and between sets of operation names (the sort mapping and the operation mapping). The operation mapping has to protect the functionality associated with an operation name, i.e. the target functionality is the image of the source functionality under the sort mapping (see also definition 4.2.-2).

A map object definition encloses at most the following clauses:

map-header (m):
introduces the name of the map object. The naming conventions for maps enforce the indication of a source and target specification object as part of the map object name. The sets of sort names of the source and

of the target object are taken as source and target of the sort mapping, while the sets of operation names of the source and target object are taken as source and target of the operation mapping.

is-clause (m):
possesses two possible entries: refinement or implementation. This clause serves to characterize the intention behind a map object:
- refinements restrict the set of models of cd.
- implementations establish a semantical connection between the source and the target specification.

base-clause (o):
the base clause offers the possibility to exclude objects of the source object hierarchy from being modified by the signature morphism. The sorts and operations of objects listed in the base clause are mapped identically.

use-clause (o):
a list of map object names. Via the use clauses, map objects may constitute hierarchies and therefore allow the incorporation of already defined map objects in a new one. Semantically, every used map represents a part of the signature morphism induced by the map object definition.

sorts-clause (m: The sorts clause of the source-object is non-empty): the sorts clause consists of pairs "old = new" where 'old' is a sort name of the source object sort name set, and 'new' is a sort name of the sort name set of the target object hierarchy. Every old sort name has to be associated to a new name.
The sort clause represents (a part of) the sort mapping induced by the map object.

ops-clause (m: the ops clause of the source object is non-empty): the ops clause consists of pairs "old = new", where 'old' is an operation name of the operation name set of the source object, and 'new' is an operation name of the operation name set of the target object hierarchy. Every old sort name has to be associated to a new name under preservation of the functionality.
The ops-clause represents (a part of) the operation mapping induced by the map object.

This brings the overview on ASPIK map objects to its end.

2.3.-2 Example: Map object STACK--M1-->QUEUE
Let QUEUE be as in 2.3.-1.
Let STACK denote a specification object of the well-known structure with sort 'stack' and operations 'push', 'pop', 'top' and 'emptystack'.
Let ELEM1 and INTEGER denote used specifications of STACK and QUEUE rep., and ELEM1--M0-->INTEGER an already defined map object.

```
map STACK--M1-->QUEUE
```

```
   is implementation
   base BOOL
   use ELEM--M0-->INTEGER
   sorts stack = queue
   ops push = enter
       pop = remove
       top = last
       emptystack = emptyqueue
endmap
```

Remarks: a) The is-clause indicates the existence of (at least) one imp object (see below).
b) BOOL is left unchanged by the signature morphism induced by STACK--M1-->QUEUE.
c) The explicit definition of ELEM--M0-->INTEGER is omitted here.

¤

Closely connected to map objects are ASPIK imp objects. Since maps only constitute syntactical relations between ASPIK specifications this would not suffice to establish 'deeper' semantical propositions (as for example the implementation concept). For this reason maps can be equipped with imp objects that provide the necessary information. Since ASPIK imp objects are still under research and outside the scope of this paper, we do not go into further details. If map and imp objects are associated to describe an ASPIK implementation, they are also called 'implementation signature' and 'implementation specification' resp. to indicate their semantical purpose.

Comming back to the branching point 'spec-terms', we can now introduce this notion. A spec-term is syntactically a specification identifier (the domain) followed by a list of map object identifiers (e.g. STACK {ELEM1--M0-->INTEGER}). Semantically, a spec-term describes a specification object that is derived from the domain hierarchy by exchanging objects, operations and sorts according to the listed signature morphisms (e.g. the specification (hierarchy) behind STACK {ELEM1--M0-->INTEGER} has all occurrences of object identifier ELEM1, sort elem1 and operations of ELEM1 substituted by occurrences of object identifier INTEGER, sort integer and INTEGER operations resp.). This use of map objects represents what is called the parameterization-by-use concept of ASPIK: specifications are not fixed structures, they show generic properties. Every (even indirect) used specification is viewed as a possible 'formal parameter' which might be actualized inside a spec-term. An ASPIK spec-term is comparable with procedure or function calls in imperative languages, where formal parameters are substituted by actual ones. Now this concept is applied to type-similar structures (specifications). Spec-terms represent a very convenient and flexible feature for re-using specifications in various contexts. For example, if there exist map objects ELEM1--M0-->INTEGER, ELEM1--M1-->NATURAL_NUMBER, one can easily use different actualizations of STACK in a new specification in parallel:

```
use STACK {ELEM1--M0-->INTEGER},
    STACK {ELEM1--M1-->NATURAL_NUMBER}
```

We close this section on the main language features of ASPIK with two remarks:

a) ModPascal provides an analogon to maps in form of instantiation definitions, and to spec-terms in form of instantiate types.
b) Most of the introduced portions of ASPIK are implemented. The software tool SPESY is described in [Sch 85]. A full language description may be found in [BV 85].

2.4. The Procedural Programming Language ModPascal

This section covers the Pascal-extension ModPascal ([Olt 84a,b]), the language of the concrete level of our assumed scenario. ModPascal encloses standard Pascal [ISO 7186], and since the latter language may be seen as a subset of ADA, a huge part of ModPascal programs may be directly expressed in ADA. The new (and different from ADA) concepts are:

- module type definitions
- enrichment definitions
- instantiation definitions
- instantiate type definitions

At first view, module types are similar to packages. But modules possess an important property that is necessary in a software development environment as well as for programming with abstract data types in general: they can be incarnated, and variables declared of a module type may be used according to the same rules that hold for ordinary types. Also, the interface of a module type is designed on theoretically insights of abstract data type theory and therefore excludes 'unclean' features.

A ModPascal module type definition consists of the following parts:

modid	:	Name of the current module.
use	:	List of modules that are used by modid.
public	:	Names and arities of new interface procedures, functions and initials.
local	:	Consists of local types, local procedures and functions, and local variables. Local items are only visible in the current module definition.
operations	:	Complete definition (name, block) of all public and local operations.

(The concrete syntax of ModPascal modules is slightly different; it was abbreviated here. See also example 2.4.-1 below.)

Modules are assumed to have an internal state. This state may be changed by the invocation of a module procedure, inspected by an module function invocation or initialized by a call of an initial operation. The mandatory local variables of a module contain the actual state in the actual bindings. Therefore, if one looks at the (abstract) data type (an algebra) described by a module definition, its set of carriers contains as single new element the cartesian product carrier of those carriers associated to the types of the local variables, and its set of operations has as new elements (the semantics of) all public operations of the current definition. The algebra constructed in this way is also called the module algebra.

Local types and local procedures and functions do not have a comparable impact on the module semantics as the local variables. The types and operations are thought to ease the programming process for the public operations. The definition blocks in the operations part may employ all features of standard Pascal and all ModPascal extensions except that nested module type definitions are not allowed. If a program requires such a structure it has to be modelled by using of previously defined modules.

2.4.-1 Example

```
type MSTACK =
     module use MELEM, MINTEGER;
       public procedure mpush(e: MELEM);
              procedure mpop;
              function mtop: MELEM;
              initial mempty;
       local type A = array[1:10] of MELEM;
              var a:A, i:MINTEGER; localend;
       procedure mpush;
              begin if i < 10
                     then begin i := i+1; a[i] := e end
                     else error                      end
       procedure mpop;
              begin if i=0 then error
                     else i := i-1                   end
       function mtop;
              begin if i=0 then error else mtop := a[i] end
       initial mempty;
              begin i := 0 end;
modend
```

This example shows a ModPascal version of bounded stack. MELEM and MINTEGER are assumed as already defined. Public operation arities omit a first parameter of type MSTACK; this parameter is supplied by the special syntax of module operation calls.

¤

The algebra carrier introduced by MSTACK is the cartesian product (A x MINTEGER) i.e. tuples of array-integer values.

The semantical operations behind mpush, mpop, etc. take these tuples as arguments and yield new tuples or select components.

Enrichments allow the extension of already defined modules by new operations. The main difference to a module type definition is, that no new data is introduced and the operations of the enrichment have to be based upon the carriers behind the enriched modules. Enrichments correspond to specifications of abstract data type with empty sort clause.

An instantiation object in ModPascal allows the hierarchical specification of signature morphisms. This is not supported in any of the existing imperative programming languages. The set of new sort names of a module consists of the one-element-set {modid} where modid ambiguously denotes the cartesian product of local variable types. The set of new operation names of a module is the set of names of all public operations with associated arities. Then, the signature morphism induced by an instantiation object links two moduls together by providing a mapping between the sort sets and an (arity-preserving) mapping between the operation sets.

Instantiation objects are used in instantiate type definitions. An instantiate type is an instance of a module or enrichment. This instance is generated according to the information provided by the instantiate type definition: a base object (module, enrichment) and a list of instantiation objects. The signature morphism induced by the instantiations is applied as substitution to the base object (hierarchy), yielding in the modified base object as instance.

Instantiations and instantiate types represent the ModPascal parameterization concept for modules (types). By this, it is not necessary to fix the parameters and non-parameters of a given module or enrichment. All substructures occuring in the source of the signature morphism are parameters in the application at hand; other applications based on other signature morphisms may select other substructures of the base object. The task of parameterization of types is left as flexible as possible for the programmer, while simultaneously a strong and theoretically well-founded formalism is introduced. (It should be noted that this kind of parameterization is also possible for familiar structures as arrays, records etc.; see [Olt 84a,b]).

Our investigation of sequential verification will not employ ModPascal in its complete size (see sec. 4.1.).

3. Language Semantics

In this section we describe the semantics of selected constructs of our version of ASPIK and ModPascal. We restrict ourselves to those topics that are of interest in the connection context between the languages.
Since connections will be established between specifications and modules/enrichments, their semantics is defined below. This includes also a definition of ASPIK maps because they may occur in the use-clause of a specification object, and, for consistency, also a definition of the ModPascal instantiation and instantiate type features.

An exhaustive presentation of the semantics can be found in [BV 85] (ASPIK) and [Olt 84b] (ModPascal).

In section 3.1. an abstract syntax describes the two language subsets to be considered in the sequel. The discussion of the semantics is splitted into two parts: the context-sensitive conditions and the dynamic semantics. The first part is given in section 3.2., whereas the second part covers sections 3.3. (semantic domains) and 3.4. (semantic clauses).

3.1. Abstract Syntax

A convenient way to describe the syntax of the ASPIK and ModPascal portions of interest is by Vienna Definition Language (VDL, [Weg 72]). We briefly introduce VDL by repeating the main notions and features which are used in sec.s 3.1.1. and 3.1.2.

VDL supports the idea of abstract syntax in that sense, that no familiar language symbols as 'begin' or 'end' (i.e. the terminal vocabulary) occur in a VDL description. Instead, all objects (syntactic entities) are collected in sets, and there are selectors that allow manipulation of them. Objects are separated into two kinds:

- elementary objects: objects with no components and therefore no selectors,
- composite objects : objects which are be composed of objects by construction operators. The components may be elementary or composite objects, and each is selectable by a unique selector.

Notation: $\{o_1, o_2\}$ denotes a set of elementary objects.
$(s_1: C_1, s_2: C_2)$ denotes a set of composite objects with selectors s_1, s_2 and component object sets C_1, C_2.

Composite objects represent tree structures in which the arcs are labelled by selectors, the leaf nodes are elementary objects and all other nodes are composite objects.

There is a distinguished elementary object, the so-called null

object $\perp$ which is different from every other elementary object. The null object is ambiguously used to denote empty domains as well as erroneous manipulations on domains.

3.1.-1. Def.: [selector application]
Let $C = (s_1: C_1, \ldots, s_n: C_n)$ denote a composite object. Let s denote a selector, and let $c \in C$ with $c = (c_1, \ldots, c_n)$.
Then (s c) is called selector application with

$$(s\ c) := \underline{case}\ s = s_i,\ i \in (n) : c_i \quad \underline{otherwise}\ \perp$$

¤

Notation: $(s^n\ c) := (s\ (s\ (s \ldots (s\ c) \ldots)$ [n times, $n > 0$]
$(s^0\ c) := c$

Selectors may be composed, too. If $(s_1: C_1)$ and $C_1 \equiv (s_2: C_2, s_3: C_3)$ are composite objects then $s_3 s_1$ is a composite selector. If $x \in (s_1: C_1)$ then $s_3 s_1$ can be applied to x to select the c_3-component.

Notation: If $s_n s_{n-1} \ldots s_1$ denotes a composite selector, then $(s_n\ (s_{n-1}\ (\ldots\ (s_1\ x) \ldots)$ denotes the application to a composite object x.

3.1.-2. Def.: [admissability]
Let $s := s_n \ldots s_1$ denote a composite selector, C a set of composite objects.
1) The application of s to $c \in C$, i.e.
$(s_n\ (s_{n-1}\ (\ldots (s_1\ c) \ldots)$
is admissable, if
$\forall\ i \in (n)\ .\ s_i\ (s_{i-1}\ (\ldots (s_1\ c) \ldots) \neq \perp$.
s is also called admissable selector for c.
2) $AD(c) := \{s \mid s$ is admissable selector for c$\}$

¤

The following conventions and operators are used:

1) We assume all object sets to be flat domains (see [Olt 84b]).

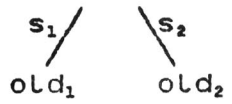
2) Syntactic Domains are denoted by identifiers starting with capital letter. Selectors and syntactic domains may occur postfixed by 'L' (for 'list'). This implies the following list structure:

DomainL = (first: Domain, rest: DomainL)

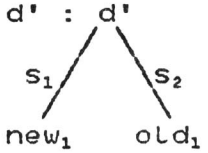
If Domain = Dom1 $\cup$ Dom2
then DomainL = Dom1L $\cup$ Dom2L

3) The L-version of a domain is not explicitly mentioned in the abstract syntax of ModPascal.
Special case: DomLL = (first: DomainL, rest: DomainLL)

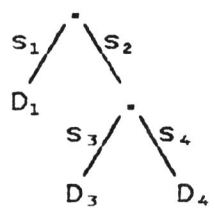
4) The general assignment operator is μ:
For $d \in D$: d
/ \



$\mu(d;\ s_1 : new_1) := d' : d'$



5) The general construction operator is μ_0:
$\mu_0(s_1 : D_1, s_2 : \mu_0(s_3 : D_3, s_4 : D_4))$ describes the domain:



The assignment and construction operators will be used in semantic clauses of sec. 3.4.

6) Below, domains of structure
(Domain* x Domain)
will be used. Sometimes it is necessary to transform elements thereof into lists:
makeList: (Domain* x Domain) ⟶ DomainL
is defined by
makeList($(d_1 \ldots d_n, d)$) := μ_0(first: d_1, rest:
μ_0(first: d_2, rest: μ_0(...(first: d_n,
rest:μ_0(first:d, rest: $\perp$) ...)

7) An operator length: DomainL ⟶ $\mathbb{N}$ that returns the number of list elements is defined for every domain. Length($\perp$) = 0.

Sometimes it is convenient to convert lists into sets:
$\mathcal{S}$: DomainL ⟶ $\mathcal{P}$(Domain)
with
$\mathcal{S}(d) := \{d_i \mid i \in (length(d)) \text{ and } \hat{\exists}\, i \in (d)\ .\ d_i = (first(rest^{i-1}(d))\}$

The operator
concat: DomainL x Domain ⟶ DomainL
is defined as expected:
concat(d, d_1) := if (first d) = $\perp$ then d_1 else
if (rest d) = $\perp$) then
μ_0(first: d, rest: μ_0(first: d_1,
rest: $\perp$))
else μ_0(first: (first d),
rest: concat((rest d), d_1))

Also
concatL: DomainL x DomainL ⟶ DomainL with
concatL(d_1,d_2) := if (first d_2) = $\perp$ then d_1 else

```
if (rest d2) = ⊥
then concat(d1, (first d2)) else
   concatL(concat(d1, (first d2)),
           (rest d2))
```

3.1.1. Specifications and Maps

An ASPIK specification object is described syntactically by the following abstract grammar:

```
Spec     = (sp_head: Sp_head, sp_body: Sp_body)
Sp_head  = (spec_id: Id, useL: Sp_termL, sorts: IdL,
            ops: OpL)
Op       = (op_id: Id, arity: Arity)
Arity    = Id* x Id
Id       = {<alphanumeric identifier>}
Sp_body  = (cons: IdL, aux: OpL, def_aux: Op_defL,
            def_car: Op_defL, def_con: Op_defL,
            priv: OpL, def_ops: Op_defL)
Op_def   = (op_head: Op_head, op_body: Op_body)
Op_head  = (op_id: Id, params: IdL)
Op_body  = Term v Case v Cond v Let
Term     = (op_id: Id, termL: TermL)
Case     = (case_var: Id, cases: CasesL)
Cases    = (tag: Term1, exit: Op_body)
Term1    = Term v {OTHERWISE}
Cond     = (if: Term, then: Op_body, else: Op_body)
Let      = (let_var: Id, let_term: Term, let_body: Op_body)
```

An ASPIK map object is described by:

```
Map      = (map_id: Map_id, is: Is, base: Sp_termL,
            use: IdL, sorts: AssocL, ops: AssocL)
Map_id   = (from: Id, to: Id, map_name: Id)
Is       = {refinement, implementation}
Assoc    = (old: Id, new: Id)
```

An ASPIK specification term (specterm) is described by

```
Sp_term   = Id v Spec_term
Spec_term = (spec_id:Id, mapL: Map_idL)
```

Note that the employed concept of an ASPIK specification is not completely coincident with [BV 85]. There a distinction is made, if for s ∈ Spec, (sp_body s) is defined or not. In the former case the specification is called algorithmic, otherwise axiomatic. Also predicate calculus formulae can be attached to specifications. For reasons described in sec. 4.1. we only consider - in terms of [BV 85] - algorithmic specifications without attached predicate calculus formulas.

3.1.2. ModPascal

In this section we introduce the abstract syntax of module type definitions, enrichment definitions, instantiation definitions and instantiate type definitions. Not every syntactic domain will be refined to full detail; see [Olt 84b] for the complete abstract syntax of ModPascal.

A ModPascal module type is defined by:

```
Module_type
          = (useL: IdL, publicL: PublicL, local: Local,
             operationL : OperationL)
Public    = Proc_head v Func_head v Init_head
Proc_head = (proc_id: Id, paramL: ParamL)
Param     = (idL: IdL, type: ID)
Func_head = (func_id: Id, paramL: ParamL, result: Id)
Init_head = (init_id: Id, paramL: ParamL)
Local     = (local_typeL: Local_typeL, local_varL: VarL,
             local_operationL: Local_operationL)
Local_type
          = Simple_type v Array_type v Record_type v
             Set_type v File_type v Pointer_type
Var       = (idL: IdL, type: Type, init: Init_stmt)
Local_operation
          = Proc_head v Func_head
Operation = Proc_spec v Func_spec v Init_spec
Proc_spec = (proc_id: Id, body: Block)
Func_spec = (func_id: Id, body: Block)
Init_spec = (init_id: Id, body: Block)
```

A ModPascal enrichment is defined by:

```
Enrich_def = (enr_id: Id, useL: IdL, addL: AddL,
              operationL: OperationL)
Add        = (add_id: Id, publicL: PublicL)
```

A ModPascal instantiation is defined by:

```
Inst_def   = useL: IdL, obj_actL: Obj_actL,
             type_actL: Type_actL, op_actL: Op_actL)
Obj_act    = (old: Id, new: Id)
Type_act   = (old: Id, new: Id)
Op_act     = (old: Id, new: Id)
```

A ModPascal instantiate type is defined by:

```
Instantiate_type = (base_type: Id, objectL: IdL)
```

3.2. Context-sensitve Conditions

3.2.1. ASPIK

We now state for the most important domains context-sensitive conditions that define the notion of static correctness for objects of the domain. The presentation of the denotational semantics in sec. 3.4. will assume correct objects.

Let s $\in$ Spec. Then its context-sensitive correctness is defined as follows:

SP1: s correct :$\Longleftrightarrow$ (sp_head s) correct and (sp_body s) correct.
SP11: sh := (sp_head s) correct :$\Longleftrightarrow$ (spec_id sh) is unique in the environment of s and (use sh) correct and (sorts sh) correct and (ops sh) correct.
SP111: uL := (useL sh) correct :$\Longleftrightarrow$ Every used specification is correct (and visible) and every used specterm is correct (and visible) and no cyclic usage of specifications occurs and all identifiers provided by the interface I_s of all used objects are unique (possibly through appropriate prefixing) (I_s = ({sort names}, {operation names})).
SP112: soL:=(sorts sh) correct : $\Longleftrightarrow$ Every sort identifier is unique in $I_s \cup$ (soL, $\emptyset$)
SP113: opL:= (ops sh) correct : $\Longleftrightarrow$ Every operation identifier is unique in $I_s \cup$ (soL, opL) and every arity is correct.
SP1131: arity correct :$\Longleftrightarrow$ $\forall$ o $\in$ (ops sh) . let $(s_1\ s_2\ \ldots\ s_n, s_{n+1})$:= (arity o) in $s_i \in ((I_s)\Downarrow 1 \cup \text{soL}), i \in (n+1)$

SP12: sb:=(sp_body s) correct :$\Longleftrightarrow$ (cons sb) correct and (aux sb) correct and (def_aux sb) correct and (def_car sb) correct and (def_con sb) correct and (priv sb) correct and (def_ops sb) correct.
SP121: coL:=(cons sb) correct :$\Longleftrightarrow$ Every identifier is contained in $\{id \mid \hat{\exists} i \in \text{length(opL)} \mid id=(\text{op\_id (first (rest}^{i-1}\ \text{opL)))}\}$
SP122: auL:=(aux sb) correct :$\Longleftrightarrow$ Every operation

identifier is unique in $I_s \cup$ (soL, opL $\cup$ auL) and SP1131 holds for every o $\in$ (aux sb)

SP123: dauL:=(def_aux sb) correct :$\Longleftrightarrow$
Every operation definition odef $\in$ dauL is correct with admissable identifier set AIS:=$I_s \cup$ (soL, opL $\cup$ auL).
For every element of auL there is an operation definition in dauL; no other definitions occur in dauL.

SP1231: opdef $\in$ Op_def correct with AIS :$\Longleftrightarrow$
(op_head opdef) is correct with AIS and
(op_body opdef) correct with AIS $\cup$
($\emptyset$, (params (op_head opdef))).

SP12311: oph:= (op_head opdef) correct with AIS :$\Longleftrightarrow$
(op_id oph) $\in$ AIS$\downarrow$2 $\setminus$ $I_s\downarrow$2 and
(params (op_head opdef)) are not contained in AIS

SP12312: opb:=(op_body opdef) correct with AIS :$\Longleftrightarrow$
Let z $\in$ {Term, Case, Cond, Let} in
case opb $\in$ z : opb is z-correct with AIS.

SP123121: opb Term-correct with AIS :$\Longleftrightarrow$
(op_id opb) $\in$ AIS$\downarrow$2 and every element of (termL opb) is Term-correct with AIS.

SP123122: opb Case-correct with AIS :$\Longleftrightarrow$
(case_var opb) $\in$ AIS$\downarrow$2 and every c $\in$ (cases opb) is correct.

SP1231221: c $\in$ (cases opb) correct :$\Longleftrightarrow$
Let V denote the new variables of (tag c) in
(tag c) is Term-correct with AIS $\cup$ ($\emptyset$, V) and (exit c) is correct with AIS $\cup$ ($\emptyset$, V).

SP123123: opb Cond-correct with AIS :$\Longleftrightarrow$
(if opb) is Term-correct with AIS and
(then opb) is correct with AIS and
(else opb) is correct with AIS.

SP123124: opb Let-correct with AIS : 222
(let_term opb) is Term-correct with AIS $\cup$ is correct with AIS $\cup$ ($\emptyset$, (let_var opb)).

SP124: dcaL := (def_car sb) correct :$\Longleftrightarrow$ For every s $\in$ soL there is a carrier predicate definition in dcaL; no other definitions occur.
Let caL := {id| $\hat{\exists}$ i $\in$ (length (dcaL)) .
(op_id (op_head (first (rest^{i-1} (dcaL))))) = id} in
Every operation definition is correct with AIS := $I_s \cup$ (soL, opL $\cup$ auL $\cup$ caL)

SP125: dcoL := (def_con sb) correct :⟺ For every c ∈ coL there is an operation definition in dcoL; no other definitions occur. Every operation definition is correct with AIS := I_s ∪ (soL, opL ∪ auL ∪ caL)
SP126: prL := (priv sb) correct :⟺ Every operation identifier is unique in I_s ∪ (soL, opL ∪ auL ∪ caL). Every arity is correct.
SP127: dopL := (def_ops sb) correct :⟺ For every operation of (opL ∪ prL) \ coL there is an operation definition in dopL; no other definitions occur. Every operation definition is correct with AIS := I_s ∪ (soL, opL ∪ prL ∪ caL)

Remarks: a) Specterm correctness (SP 111) is defined below.
b) The interface I_s of a specification is the valid name space generated by all sort and operation identifiers of (transitively) used specifications. It is also called imported interface, whereas the exported interface contains additionally the sorts and (public) operations of the current specification.

Let m ∈ Map. Then its context-sensitive correctness is defined as follows:

MA1: m correct :⟺ (map_id m) is correct and (base m) is correct and (use m) is correct and (sorts m) is correct and (ops m) is correct
MA11: mid := (map_id m) correct :⟺ The specifications (from mid) and (to mid) are visible and correct and (map_name mid) is unique in the environment of m.
MA12: baL := (base m) correct :⟺ Every specterm in baL is correct.
MA13: uL := (use m) correct :⟺ Every used map object is correct and the union of all mappings induced by used objects is itself a signature morphism.
MA14: soL := (sorts m) correct :⟺ For every association a ∈ soL it holds: (old a) is visible sort identifier in the hierarchy spanned by (from mid) and (new a) is visible sort identifier in the hierarchy spanned by (to mid) and (new a) as well as (old a) are not sort identifier of some specification of baL.
MA15: opL := (ops m) correct :⟺ For every association

a ∈ opL it holds: (old a) is visible operation identifier in the hierarchy spanned by (from mid) and (new a) is visible operation identifier in the hierchy spanned by (to mid) and (new a) as well as (old a) are not operation identifier of some specification of baL and the functionalities of (old a) and (new a) are compatible (i.e. the signature morphism property is satisfied).

Remarks: a) Specterm correctness (MA12) is defined below.
b) Since different used objects may involve the same object, and therefore sort and operation mappings are defined on the same source sets it has to be guaranteed in MA13 that in this case equal arguments yield equal results (i.e. the function property of the union of all used signature morphisms).

Let st ∈ Sp_term. Then its context-sensitive conditions are:

ST1: st correct :⟺ case st ∈ Id : st denotes a correct specification; case st ∈ Spec_term : (spec_id st) denotes a correct specification and (mapL st) is correct and (mapL st) is applicable to (spec_id st)
ST2: mL := (mapL st) correct :⟺ every element of mL is a correct mapobject and the union of all elements of mL is a signature morphism
ST3: mL is applicable to (spec_id st) :⟺ let mp := the signature morphism induced by mL, sh := the hierarchy spanned by (spec_id st) in source(mp) ⊆ sh

Remarks: a) The notion of 'union of signature morphisms' (ST2) stands for union of source and target sets of sort and operation mappings where the arity operators for each operation name set are maintained.

3.2.2. ModPascal

In the following context-sensitive conditions will be given for the object domains introduced in sec. 3.1.2. The full set of conditions for ModPascal and more details may be found in [Olt 84a].

Let m ∈ Module_type. Then its context-sensitive correctness is defined as follows:

MU1: m correct :$\Longleftrightarrow$ (useL m) correct and (publicL m) correct and (local m) correct and (operationL m) correct and m interface correct

MU11: usL := (useL m) correct :$\Longleftrightarrow$ Every used module or enrichment is visible and correct and no cycles in the use-relation occur and all identifiers provided by the interface I_M of all used objects are unique (possibly through appropriate prefixing).

MU12: puL := (publicL m) correct :$\Longleftrightarrow$ For every element oph of puL it holds:
case oph $\in$ Proc_head : (proc_id oph) and (paramL oph) contain unique identifiers and $\forall$ i $\in$ (length((paramL oph))) . (type (first (rest $^{i-1}$ (paramL oph)))) $\in$ ($I_M\downarrow 1 \cup$ {m_id})
case oph $\in$ Func_head : (func_id oph) and (paramL oph) and (result oph) contain unique identifiers and $\forall$ i $\in$ (length((paramL oph))) . (type (first (rest $^{i-1}$ (paramL oph)))) $\in$ ($I_M\downarrow 1 \cup$ {m_id}) and (result oph) $\in I_M\downarrow 1$
case oph $\in$ Init_head : (init_id oph) and (paramL oph) contain unique identifiers and $\forall$ i $\in$ (length((paramL oph))) . (type (first (rest $^{i-1}$ (paramL oph)))) $\in I_M\downarrow 1$
and at least one initial header occurs.

MU13: lp := (local m) correct :$\Longleftrightarrow$ (local_typeL lp) correct and (local_varL lp) correct and local_operationL lp) correct and length ((local_varL lp)) > 0

MU131: ltL := (local_typeL lp) correct :$\Longleftrightarrow$ no introduced type is a module type and all type identifiers are unique and all occurring type identifiers are either defined in ltL or contained in $I_M\downarrow 1$

MU132: lvL := (local_varL lp) correct :$\Longleftrightarrow$ all variable names are unique and all variable types are either contained in $I_M\downarrow 1$ or are implicit non-module types

MU133: loL := (local_operationL lp) correct :$\Longleftrightarrow$ For every element oph of loL it holds:
case oph $\in$ Proc_head : (proc_id oph) and (paramL oph) contain unique identifiers and $\forall$ i $\in$ (length((paramL oph))) . (type (first (rest $^{i-1}$ (paramL oph)))) $\in$ ($I_M\downarrow 1 \cup$ {m_id} $\cup$ {<type identifier of ltL>})
case oph $\in$ Func_head : (func_id oph) and (paramL oph) and (result oph) contain

unique identifiers and
∀ i ∈ (length((paramL oph))) .
(type (first (rest $^{i-1}$ (paramL oph)))) ∈
(I_M↓1 ∪ {m_id} ∪ {<type identifier of ltL>})
and (result oph) ∈ (I_M↓1 ∪ {m_id} ∪
{<type identifier of ltL>})

MU14: opL := (operationL m) correct :⟺ For each public and each local operation heading of puL and loL there is exactly one operation definition in opL and no other definition occurs and all operation definitions of opL are correct.

MU141: opd ∈ opL is correct :⟺ let I_1 := I_M ∪
({m_id, <local type identifiers>}, {<operation identifiers of puL and loL>}), I_2 := I_1 \
(∅, {<initial operation identifiers of puL>}),
lv := {<variable identifier of lvL>},
fp := {<formal parameters of opd in puL>},
V_1 := lv ∪ fp in
case opd ∈ Proc_spec : (body opd) is correct with I_2 and V_1
case opd ∈ Func_spec : (body opd) is correct with I_2 and V_1
case opd ∈ Init_spec : let I_3 := I_1 \
(∅, {<procedure and function identifiers in puL>}) in
(body opd) correct with I_3 and V_1;
and global variables of (body opd) are contained in lv

MU1411: bd := (body opd) correct with I and V :⟺ all (free) type identifiers of bd are contained in I↓1 and all (free) operation identifiers of bd are contained in I↓2 and all (free) variable identifiers are contained in V and bd is block-correct.

MU14111: (body opd) block-correct :⟺ <see [Olt 84a] for the correctness of blocks and remark c) below>

Remarks: a) The module identifier m_id is associated to the embedding type definition domain Type_def (see [Olt 84b], sec. 2.1.2). Its correctness is assumed.

b) The interface I_M = (OB, OP) of a module M is the valid name space generated by all module (OB) and module operation (OP) names of (transitively) used objects. It is also called imported interface, whereas the exported interface contains additionally the module name and module operation names of the current module M.

c) Contrary to the concrete ModPascal syntax of [Olt 84a] we here assume no implicit parameters of procedures, functions and initials. Therefore

(paramL oph) in MU12 selects also the (first-position) module argument (see also sec. 3.2.2., CM2 in [Olt 84a]).

d) The correctness of blocks is coincident with Pascal context-sensitive conditions for blocks except the restrictions that the set of global variables is restricted to the set of local variables of the module and that no nested module type definitions occur.

Let e ∈ Enrich_def. Then its context-sensitive correctness is defined as follows:

EN1: e correct :⟺ (enr_id e) is unique in the environment of e and (useL e) is correct and (addL e) is correct and (operationL e) is correct
EN11: uL := (useL m) correct :⟺ Every used module or enrichment is visible and correct and no cycles in the use-relation occur and all identifiers provided by the interface I_E of all used objects are unique (possibly through appropriate prefixing).
EN12: aL := (addL e) correct :⟺ all operation identifiers introduced in aL are distinct and for all elements ad of aL it holds: (add_id ad) ∈ {id| id ∈ $I_E\downarrow 1$ ∧ id is module name} and (publicL ad) is nonempty and (publicL ad) is correct in the environment of (add_id ad)
EN121: (publicL ad) correct in (add_id ad) :⟺ (publicL (add_id ad)) ∪ (public ad) is correct <see MU12>
EN13: oL := (operationL e) is correct :⟺ For each public operation heading of each element ad of aL there is exactly one operation definition in oL and no other operation definition occurs and all operation definitions are correct
EN131: opd ∈ oL is correct :⟺ let mod := (add_id ad) where ad denotes the element of aL in which the associated operation header is defined in let I_1 := I_E ∪ (∅, {<operation identifiers defined in aL>}), lV := {<local variable identifier of mod>}, fp := {<formal parameters of opd in aL>}, V_1 := lV ∪ fp in case opd ∈ Proc_spec or opd ∈ Func_spec : (body opd) is correct with I_1 and V_1 case opd ∈ Init_spec : let I_2 := I_1 \ (∅, {<procedure and function identifier in (publicL ad)>}) in

(body opd) is correct with I_2 and V_1; and global variables of (body opd) are contained in lv.
EN1311: (body opd) correct with I and V :⟺ <see MU1411>

Remarks: a) The interface I_E = (OB, OP) of an enrichment E is the valid name space generated by all module (OB) and module operation (OP) names of (transitively) used objects. It is called imported interface whereas the exported interface contains additionally the module operation names of the current enrichment E.

Let i ∈ Inst_def. Then its context-sensitive correctness is defined as:

ID1: i correct :⟺ (inst_id i) is unique in the environment of i and (useL i) correct and (obj_actL i) correct and (type_actL i) correct and (op_actL i) correct and at least either the uselist or some actualization list are nonempty and (useL i) and the actualizations together describe a signature mophism (see remark a) below)
ID11: uL := (useL i) correct :⟺ Every used instantiation is visible and correct and no cycles in the use-relation occur and the union of all used objects describes a signature morphism.
ID12: oL := (obj_actL i) correct :⟺ For each element ob of oL it holds: (new ob) and (old ob) are either both module or enrichment identifiers and (new ob) and (old ob) are visible and correct.
ID13: tL := (type_actL i) correct :⟺ For every element tp of tL it holds: (new ob) and (old ob) are both module identifier and (new ob) and (old ob) are occurring in enrichments that constitute an object actualization element of oL.
ID14: pL := (op_actL i) correct : 222 For every element op of pL it holds: (new op) and (old op) are public operation names of objects (new ob) and (old ob) of some element ob of oL and the associated functionalities obey the signature morphism property (see remark a) below)

Remarks: a) The concept of signature morphism is very crucial

in this context. A first definition is given below, whereas its Definition: 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 $A(op)=ob_1 ob_2 \ldots ob_n$, then $ob_1 \ldots ob_{n-1}$ 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 $A_1(op)=ob_1 \ldots ob_n$. $A_2(g(op))=f(ob_1)\ldots f(ob_n)$

¤

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.

Let i ∈ Instantiate_type. Then its context-sensitive correctness is defined by:

IT1: i correct :⟺ (base_type i) is a module or enrichment object and (objectL i) is correct and (objectL i) is applicable to (base_type i)
IT11: oL := (objectL i) correct :⟺ Every element of oL is a visible and correct instantiation object and let sig denote the union of all instantiation objects of oL in sig is a signature morphism
IT12: oL is applicable to (base_type i) :⟺ all source objects of sig are contained in the hierarchy spanned by (base_type i)

Remarks: a) The instantiate type identifier is associated to the embedding type definition domain Type_def (see [Olt 84b], sec. 2.1.2.). Its correctness is assumed.
b) The applicability is defined stronger in sec. 3.7.2. of [Olt 84b] (operator Comp?) such that hierarchical conditions are respected. For the purposes of this paper IT12 suffices.

3.3. Semantic Domains and Semantic Functions

The semantic domains introduced in this section are choosen such that a concise and sufficient description is possible of both languages as well as of the correctness concept of section 4. Therefore the number of domains is increased compared to the case of a single language semantics. On the other hand, there are domains that will serve for the semantics definition of ASPIK and ModPascal, and are also important in sec. 4 (e.g. the domain Alg of strict algebras). This is intended since it facilitates the comparison of ASPIK and ModPascal structures. The set of domains is based on [Olt 84b].

Section 3.3.2. deals with semantic functions. Also there are some additions and modifications compared to [Olt 84b] since the treatment of ASPIK constructs requires different functionalities. Details of ModPascal related semantic functions are omitted here; details of the construction of the central domain Alg are postponed (serving as target domain for all object definitions; see sec. 4.2.).

ASPIK, as defined in [BV 83] and [BV 85], was originally supplied with a category-theoretic semantics. For the purpose of this paper and our modified version of ASPIK the semantics has been reformulated in terms of a denotational semantics compatible with those of ModPascal. This also influenced the set of employed domains.

3.3.1. Domains

The following semantic domains are used in the semantic clauses of sec. 3.4.:

Flat Domains:

D_BOOL
= {true, false}: The boolean values.

D_INT
= { ..., -2, -1, 0, 1, 2, ...}: The integer values.

Id
= {id| id ∈ {A, ..., Z, 0, ..., 9}$^+$ ∧ first(id) ∉ {0, ..., 9}}: Identifiers are strings of letters and digits, starting with a letter.

Map
= Id ⟶ Id: serves as domain for mapping definitions by ASPIK map-objects or ModPascal instantiations.

Alg
= + {Alg[Σ]| Σ is signature}: The domain of algebras. It is constructed as the direct sum (or coalesced) sum of signature dependant algebra domains. Alg is not "the set of sets", but a set of all interesting strict algebras to describe semantics of data types; see sec. 4.2.

Loc
= {an unbound domain of locations}: If locations are interpreted as main memory addresses, Loc could be seen as integer subset. But every interpretation into distinguishable elements will work.

AlgQual
= {SPEC, MAIN, BOOLEAN, INTEGER, REAL, CHAR, SCALAR, SUBRANGE, ARRAY, RECORD, FILE, SET, POINTER, MODULE, ENRICHMENT}: The algebra qualifications indicate the basing structure for an algebra. MAIN refers to the main program algebra.

ObQual
= {CTASEL, MODSEL, SORT, REPSEL, ENRSEL, REPOB, LAB, CONST, VAR, PROC, FUNC, INIT, INST} + AlgQual: The object qualifications indicate either the basing ModPascal feature of an item or the basing ModPascal type.

ValQual
= {C| C = TOI(A)↓1 for A ∈ Alg}: All carriersets of interest for algebras in Alg. ValQual may be seen as a factorization of Alg (TOI = type of interest; see [Olt 84b]).

Arity
= (Id* x Id): provides arities (functionalities) for ASPIK operations.

ArDes
= (Id ——> Arity): technical; combines operation and arity.

SigMorph
= (Map x Map x ArDes): Signature morphisms. The first two components contain the object and operation mapping resp. The third component is a set of arity associations. An exact definition is given in sec. 4.3.4.

AClauseOps
= {cta?, use?, sorts?, p_op_id?, p_op_ar?, constr?, aux_id?, aux_op_ar?, car_def?, pr_op_id?, pr_op_ar?, cons_def?, sop_def?}: The set of predefined identifiers for syntactic operators on specifictions. They are connected to the notion of cta-environment (see definition 3.4.1.-1).

MClauseOps
= {malg?, muse?, p_proc_id?, p_func_id?, p_init_id?, l_type_id?, l_proc_id?, l_func_id?, l_var_id?, p_proc_ar?, p_func_ar?, p_init_ar?, l_proc_ar?, l_func_ar?, l_var_type?, map_def?}: The set of predefined identifiers for syntactic operators on module type definitions. They are connected to the notion of mod-environment (see definition 3.4.2.-1).

EClauseOps
= {enr?, euse?, add_id?, add_proc_id?, add_func_id?, add_init_id?, add_proc_ar?, add_init_ar?, add_func_ar?, cop_def?}: The set of predefined identifiers for syntactic operators on enrichment definitions. They are connected to

the notion of enr-environment (see definition 3.4.2.-2).

RClauseOps
= {rob?, ruse?, connect?, operations?, rf_ar?, rf_def?}: The set of predefined identifiers for syntactic operators on rep-objects (see sec. 4.3.). They are connected to the notion of rep-environment (see definition 4.3.4.-1).

Not necessarily flat domains:

Store
= (Loc ⟶ Val): links locations and values.

Env
= (Id ⟶ (Loc x ObQual x ValQual)): Each identifier id ∈ Id is connected to a triple. The second and third components describe properties of id.

State
= Env x Store : Characterization of a state as tuple. See also the memory model in sec. 3.3.3.

Trans
= (State ⟶ State): State transformation that are induced by programming language constructs will be described with T ∈ Trans.

ETrans
= (State ⟶ (State x Val)): Analogously Trans, but with values out of Val.

OpDen
= $\dot{+}_{n,m\in\mathbb{N}}$ (ValQual^n ⟶ ValQual^m): Function between n-ary and m-ary cartesian products of ValQual. A generalization of functions of algebras of Alg.

Val
= D_BOOL + D_INT + Id + Alg + ValQual + OpDen

OpDes
= (Id x OpDen): technical; combines operation and denotation.

```
D_BOOL = {true, false}
D_INT = {..., -1, 0, 1, ...}
Id = {id| id ∈ {A, ..., Z, 0, ..., 9}+ ∧ first(id) ∉
                                          {0, ..., 9}}
Map = Id ⟶ Id
Alg = + {Alg[Σ]| Σ is signature}
Loc = {unbound domains of locations}
AlgQual = {SPEC, MAIN, BOOLEAN, INTEGER, REAL, CHAR,
           SCALAR, SUBRANGE, ARRAY, RECORD, FILE, SET,
           POINTER, MODULE, ENRICHMENT}
ObQual = AlgQual + {CTASEL, MODSEL, REPSEL, ENRSEL, SORT,
                    REPOB, LAB, PROC, FUNC, VAR, INIT}
```

```
ValQual = {C | C = TOI(A)↓1 for A ∈ Alg}
Val = D_BOOL + D_INT + Id + Alg + ValQual + OpDen
Store = Loc ——> Val
Env = Id ——> (Loc x ObQual x ValQual)
State = Env x Store
Trans = State ——> State
ETrans = State ——> (State x Val)
OpDen = + {Val^n ——> Val^m | n, m ∈ ℕ}
Arity = (Id* x Id)
ArDes = (Id ——> Arity)
AClauseOps =  ┐
MClauseOps =  │  (as above)
EClauseOps =  │
RClauseOps =  ┘
```

In the following we assume that the syntactic domain Id and the semantic domain Id are identical.

3.3.2. Functions

The syntactic and semantic domains are linked by the following functions, that are based on the overall domain Constr:

```
Constr = Spec + Sp_head + Op + ... +
         Module_type + Public + ...
```

(i.e. Constr is the coalesced sum of all syntactic domains used in sec. 3.1. for ASPIK and ModPascal; in the ModPascal case, all domains of [Olt 84b] are contained in Constr. Ambiguously denoted domains D are assumed to be tagged appropriately (D_A, D_M)).

c ∈ Constr:

If no exception for c is listed below, the semantic function

```
M: Constr ——> State ——> State
```

is applicable.

M links an initial state prior execution of a language construct to a state after execution of it. M is defined by the semantic clauses of sec. 3.4. which are elaborated to an appropriate level of detail.

Notation: Elements of Constr will be enclosed in double brackets ⟦ and ⟧. Elements (ς, σ) of State will be supplied to M with juxtaposed components.
Example: M⟦c⟧$\varsigma\sigma$

C ∈ Expr:

(b) E: Expr ——> State ——> (State x Val)
and M⟦c⟧$\varsigma\sigma$ ⇒ E⟦c⟧$\varsigma\sigma$

C ∈ (Stand_type ∨ Stand_type_gen):

(c) Mt: (Stand_type ∨ Stand_type_gen) ——> State ——> (ObQual x ValQual x Alg)

and M⟦c⟧ξσ ⇒ Mt⟦c⟧ξσ

c ∈ Module_type:

(d) Mm: Module_type ⟶ State
⟶ ((ObQual × ValQual × Alg) × State)
and M⟦c⟧ξσ ⇒ Mm⟦c⟧ξσ

C ∈ Enrich_def:

(e) Me: Enrich_def ⟶ State ⟶ State
and M⟦c⟧ξσ ⇒ Me⟦c⟧ξσ

C ∈ Instantiate_type:

(f) Mi: Instantiate_type ⟶ State ⟶ ((ObQual × ValQual × Alg) × State)
and M⟦c⟧ξσ ⇒ Mi⟦c⟧ξσ

In the semantic clauses for ModPascal also the following auxiliary functions occur:

newloc
newloc gets a currently unused location of an environment.

newloc: Env ⟶ Loc
newloc(ξ) := η loc . ∀ id ∈ Id . ξ(id)↓1 ≠ loc

searchdef
searchdef looks for the algebra to which an operation is associated; it returns the algebra identifier.

searchdef: Id ⟶ State ⟶ Id
searchdef(opid)ξσ :=
let id := ι id_1 ∈ Id . ξ(id_1)↓2 ∈ AlgQual and
let (C, F) := σ(ξ(id_1)↓1) in
opid ∈ opnames(F) in
id

(ι returns $\perp_{Id}$ if no unique id_1 exists with the required property)

standard
indicates whether an identifier denotes a standard object, and provides its initialization value in the positive case.

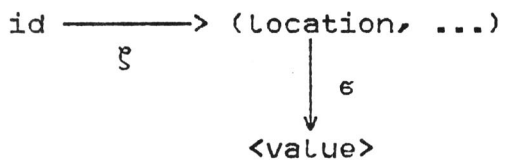
standard: Id ⟶ (D_Bool × Val)
standard(id) :=
if id = BOOL ⟶ (true, false) else
if id = INT ⟶ (true, 0) else
.
.
.
else (false, ⊥)

3.3.3. Memory Model

Though we are considering an applicative language, for reasons of compatibility an environmental view is taken in the formulation of semantical clauses for ASPIK. That means that the declaration of a spec or the evaluation of a term takes place in a given argument state; the state is modified in the case of declaration. This view is no a-priori violation of the requirements for applicative PLs, as given in sec. 2.1. If the rules are respected - as we do -, then the state-oriented model shows the same behaviour as e.g. any purely functional model that in a different way keeps track of its visible objects.

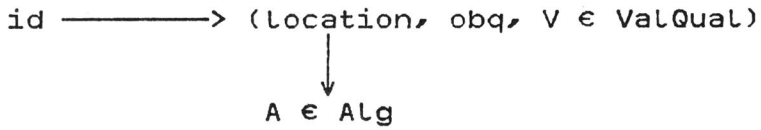
Our states follow a two-level memory model (that was primarily used for procedural PLs, but fits also for applicative PLs): The first level, represented by the domain Env of environments, links identifiers to a vector of values. One of them is a location of a (virtual) memory, in which an associated value is stored. This represents the second level of the memory model, and it is formed by the domain Store.

Using $\varsigma \in Env$, $\sigma \in State$ we have for $id \in Id$:



For modules/enrichments and spec objects we have

$obq := \varsigma(id)\downarrow 2 \in \{MODULE, ENRICHMENT, SPEC\}$



(Note the extension of the memory model in sec. 4.3.4.).

3.4. Semantic Clauses

Before we state the most important semantical equations for ASPIK and ModPascal, we introduce some notational conventions frequently occurring lateron.

Notations

$\mathbb{N}$ denotes the set of natural numbers.

For a natural number n, (n) denotes the set $\{1, \ldots, n\}$, and $[n] := (n) \cup \{0\}$.

For vectors $v = (v_1, \ldots, v_n)$, $(v_1, \ldots, v_n)\downarrow i$ or $v\downarrow i$ denotes the i-th component v_i of v.

For a set s, $\mathcal{P}(s)$ denotes the power set of s.

$\hat{\exists}$ denotes the unique existential quantification.

For a mapping m: A ⟶ B defined by m: $\subseteq$ (A x B), the substitution m[a $\mapsto$ a_1] denotes (m $\setminus$ {(a, m(a))}) $\cup$ {(a, a_1)}.

Four operators are used for functional abstraction:
- λx . term: Bounds free occurrences of x in term. This abstraction is equivalent to a definition 'F(x) = term' of a function F.
- ιx . cond :
  Bounds x in cond and qualifies the x as unique to fullfill cond. Equivalent to: $\hat{\exists}$ x . (cond = true). If no unique x exists, ι evaluates to $\perp$.
  Example: n := ιi . (i+1=5) $\Rightarrow$ (n=4)
- fix f . term:
  Bounds free occurrences of f in term and denotes the least fixpoint of the functional equation F = term[F] where term[F] is a term with free occurrences of F.
  Example: fix f . (λn . if n = 0 then 1 else n*f(n-1))
  denotes the least fixpoint of the functional equation F(n) = if n = 0 then 1 else n*F(n-1), that is the standard faculty function.

- η x . cond :
  bounds x in cond and qualifies x as one possible value that satisfies cond. Equivalent to: $\exists$ x . (cond = true). If no value exists that satisfies cond, η evaluates to $\perp$.
  Example: n := η x . (x*x = 9) $\Rightarrow$ n $\in$ {3, -3})

If indexed items occur themselves in index positions, the indices are juxtaposed in parenthesis.
Example: X_n ⟶ $Y_{X(n)}$ ⟶ $Z_{Y(X(n))}$
$X_{i,j}$ ⟶ $Y_{X(i)(j)}$

3.4.1. ASPIK

As mentioned above, this section is an abbreviated and condensed reformulation of the category-theoretic semantics of ASPIK as given in [BV 85]. For reasons described in sec. 4.1. we do not treat the semantics of axiomatic specifications; instead of we present a semantics of algorithmic spec objects in full detail which is not equivalent to [BV 85] but fits into our purposes. To adapt it to the denotational environment, some additional operators and features have to be included. They cover mostly the involvation of specific syntactic information in states and the processing of this information in the computation of specification semantics.

The first definition of this kind deals with a characterization of environments $\xi \in$ Env that provide predefined identifiers that collect information about syn-

tactic structures of specifications. This information must be gathered because our application of this semantics in sec. 4 must have access to the syntactic items that generated the specific meaning. This is similar to familiar issues of imperative programming languages as type checking or scoping, which - if modelled in a denotational setting - would require an analogous proceeding. In general, it would suffice to consider only the overall meaning of a spec object S (an algebra), and incorporate this in semantic clauses involving S.

Syntactic information is stored in special slots of environments that contain for all visible spec objects relevant values of specific clauses.

3.4.1.-1 Def. [cta-environment]

Let AClauseOps ∈ Id with
AClauseOps = {cta?, use?, sorts?, p_op_id?, pr_op_id?, constr?, aux_id?, p_op_ar?, aux_op_ar?, pr_op_ar?, car_def?, cons_def?, op_def?}.
Then ς ∈ Env is called cta-environment, if for x ∈ AClauseOps
a) ς(x) ≠ ⊥, and
b) ς(x)↓2 = CTASEL ∎

Remark: Associated to every element el of AClauseOps is a (ambiguously denoted) special function el that evaluates to syntactic information if applied to specification and operation identifiers:

el = cta?:
Associated operation: cta? : Id ⟶ State ⟶ D_BOOL
ς(cta?) = (loc, CTASEL, ⊥)
σ(loc) = {(id, tv) | id ∈ Id, tv ∈ D_BOOL}
cta?(id)ςσ := σ(ς(cta?)↓1) (id)

el ∈ {use?, sorts?, p_op_id?, pr_op_id?, constr?, aux_id?}:
Associated operation: el : Id ⟶ State ⟶ IdL
ς(el) = (loc, CTASEL, ⊥)
σ(loc) = {(id, (id$_1$, ..., id$_n$)) | id, id$_i$ ∈ Id, i ∈ (n), n ∈ ℕ}
el(id)ςσ := σ(ς(el)↓1) (id)

el ∈ {p_op_ar?, aux_op_ar?, pr_op_ar?}:
Associated operation: el: Id ⟶ Id ⟶ State ⟶ Arity
ς(el) = (loc, CTASEL, ⊥)
σ(loc) = {(id, ad) | id ∈ Id, ad ∈ ArDes}
el(id$_1$,id$_2$)ςσ := σ(ς(el)↓1)(id$_1$)(id$_2$)

el ∈ {car_def?, cons_def?, op_def?}:
Associated operation: el : Id ⟶ State ⟶ Op_defL
ς(el) = (loc, CTASEL, ⊥)
σ(loc) = {(id, (opd$_1$, ..., opd$_n$)) | id ∈ Id, opd$_i$ ∈ Op_def, i ∈ (n), n ∈ ℕ}
el(id)ςσ := σ(ς(el)↓1) (id)
∎

In a given state (ξ, σ), $\sigma(\xi(op)\downarrow 1)$ for op $\in$ AClauseOps denotes a functional relation that, applied to a specification identifier, evaluates to syntactic information about the specification (for illustration, see definition 3.4.1.-2 below).

Notation Let $R := \{(x,y) \mid x \in X, y \in Y\}$ denote a lefttotal, right unique relation on $(X \times Y)$. Then $R(x)$ denotes the application of R to $x \in X$ and $R(x) = y :\Longleftrightarrow (x,y) \in R$. If $R \subseteq (X \times Y \times Z)$, then $R(x)(y) = z :\Longleftrightarrow (x,y,z) \in R$

The next operator is technical. By application of EXT, the functional relation of an element of AClauseOps is extended (i.e. source and target of the associated operation are enlarged).

3.4.1.-2 Def. [EXT]

Let ClauseVal be defined as above.
Let $(\xi,\sigma) \in$ State with ξ cta-environment. Let

EXT: Id ⟶ Id ⟶ ClauseVal ⟶ State ⟶ State

with

EXT $(id_1, id_2, cv)\xi\sigma$:=
  if not $(id_1 \in$ AClauseOps) then $\perp$ else
  (case id_1 =
    cta?:
      if not (cv $\in$ D_BOOL) then $\perp$
    case $id_1 \in$ {use?, sorts?, p_op_id?, constr?, aux_id?, pr_op_id?}:
      if not (cv $\in$ IdL) then $\perp$ else
      let $\sigma_1 := \sigma[\xi(id_1)\downarrow 1 \hookleftarrow \sigma(\xi(id_1)\downarrow 1) \cup \{(id_2, cv)\}]$ in
    case $id_1 \in$ {p_op_ar?, aux_op_ar?, pr_op_ar?}:
      if not (cv $\in$ ArDesL) then $\perp$ else)
  let $\sigma_1 := \sigma[\xi(id_1)\downarrow 1 \hookleftarrow \sigma(\xi(id_1)\downarrow 1) \cup (\{id_2\} \times \{\wp(cv)\})$ in
    case $id_1 \in$ {car_def?, cons_def?, op_def?}:
      if not (cv $\in$ OpDesL) then $\perp$ else
      let $\sigma_1 := \sigma[\xi(id_1)\downarrow 1 \hookleftarrow \sigma(\xi(id_1)\downarrow 1) \cup (\{id_2\} \times \{\wp(cv)\})$
      in
      (ξ, σ_1)

¤

Remarks a) $\wp$ denotes the list transformation (into sets) of sec. 3.1.
b) EXT is applied in the semantic clauses below during extraction of syntactic structures. Thereafter it is possible to refer to these structures via the state.

If for a given S $\in$ Spec all AClauseOps elements should be updated in a given state, the operator EXTEND is used:

3.4.1.-3 Def. [EXTEND]
Let s $\in$ Spec, $(\xi, \sigma) \in$ State with ξ cta-environment.
Then the operator

```
  EXTEND: Spec ──> State ──> State
is defined as
  EXTEND(s)ξσ :=
   Let s_id := (spec_id (sp_head s)),
       (u_1, ..., u_n) := (use (sp_head s)),
       (s_1, ..., s_a) := (sorts (sp_head s)),
       (o_1, ..., o_b) := (ops (sp_head s)),
       (a_1, ..., a_c) := (aux (sp_body s)),
       (c_1, ..., c_d) := (cons (sp_body s)),
       (p_1, ..., p_e) := (priv (sp_body s)),
       (aud_1, ..., aud_c) := (def_aux (sp_body s)),
       (cad_1, ..., cad_a) := (def_car (sp_body s)),
       (cod_1, ..., cod_d) := (def_cons (sp_body s)),
       (opd_1, ..., opd_k) := (def_ops (sp_body s))      in

   Let (ξ_1, σ_1) := EXT(cta?, s_id, true)ξσ,
       (ξ_2, σ_2) := EXT(use?, s_id, (u_1, ..., u_n))ξ_1σ_1,
       (ξ_3, σ_3) := EXT(sorts?, s_id, (s_1, ..., s_a))ξ_2σ_2,
       (ξ_4, σ_4) := EXT(p_op_id?, s_id, (oid_i | ∃ i ∈ (b) .
                         (op_id p_i) = oid_i, i ∈ (b)))ξ_3σ_3,
       (ξ_5, σ_5) := EXT(p_op_ar?, s_id,
                         (ops (sp_head s)))ξ_4σ_4
       (ξ_6, σ_6) := EXT(pr_op_id?, s_id, (oid_i | ∃i ∈ (e) .
                         (op_id p_i) = oid_i))ξ_5σ_5,
       (ξ_7, σ_7) := EXT(pr_op_ar?, s_id,
                         (priv(sp_body s)))ξ_6σ_6,
       (ξ_8, σ_8) := EXT(aux_id?, s_id, (oid_i| ∃ i ∈ (c) .
                         (op_id a_i) = oid_i))ξ_7σ_7,
       (ξ_9, σ_9) := EXT(aux_op_ar?, s_id,
                         (aux (sp_body s)))ξ_8σ_8,  in
   Let (ξ_0, σ_0) := (ξ_9, σ_9)  in
   Let (ξ_1, σ_1) := EXT(constr?, s_id, (c_1, ..., c_d))ξ_0σ_0,
       (ξ_2, σ_2) := EXT(aux_def?, s_id, (aud_1, ..., aud_c))ξ_1σ_1,
       (ξ_3, σ_3) := EXT(car_def?, s_id, (cad_1, ..., cad_a))ξ_2σ_2,
       (ξ_4, σ_4) := EXT(cons_def?, s_id,
                         (cod_1, ..., cod_d))ξ_3σ_3,
       (ξ_5, σ_5) := EXT(op_def?, s_id, (opd_1, ..., opd_k))ξ_4σ_4
     in (ξ_5, σ_5)
                                                              ¤
```

Remark: EXTEND produces a state in which all necessary syntactic information about a spec is stored in slots defined by elements of AClauseOps.

3.4.1.-4 Def. [|F|]
Let S denote a set and BOOL = {true, false} denote the boolean values. Then, for a function F : S ──> BOOL, the extension |F| of F is defined as

$$|F| := \{s \mid s \in S \text{ and } F(s) = true\}.$$ ¤

The next definition introduces the important notion of a Herbrand universe in terms of cta-environments. Herbrand universes are sets of all well-formed terms built from given operation symbols and arities.

3.4.1.-5 Def. [H]
Let $(\xi,\sigma) \in$ State with ξ cta-environment.
Let id $\in$ Id with $\xi(id)\Downarrow 2 = SPEC$, $(s_1, \ldots, s_a) := sorts?(id)\xi\sigma$
Let $(c_1, \ldots, c_n) := constr?(id)\xi\sigma$
Let $(s_{i1} \ldots s_{im(i)}, s_{i(m(i)+1)}) := p\_op\_ar?(id, c_i)\xi\sigma$, $i \in (n)$ $m_i \in \mathbb{N}$

Then the Herbrand universe(s) $H_{s(1)}, \ldots, H_{s(a)}$ of id is (are) defined by:
For $j \in (a)$, $i \in (n)$
$H_{s(j)}$ is the smallest set with
i) if $m_i=0$ and $s_{i(m(i)+1)} = s_j$ then $c_i \in H_{s(j)}$
ii) if $h_1 \in H_{s(i1)}, \ldots, h_{m(i)} \in H_{s(im(i))}$ and $s_{i(m(i)+1)} = s_j$
then $c_i(h_1, \ldots, h_{m(i)}) \in H_{sj}$
iii) $\perp_{s(j)} \in H_{s(j)}$
The operator
H : Id ⟶ State ⟶ ValL
is defined as
$H(id)\xi\sigma := (H_{s(1)}, \ldots, H_{s(a)})$
where $(\xi,\sigma) \in$ State, ξ cta-environment and
$a := length(sorts?(id)\xi\sigma)$. ¤

Remarks a) The H image (an element of ValL) is always a sequence of sets. The case of an empty set as sequence element is excluded by the context sensitive conditions imposed on specification objects (see sec. 3.2.1., SP1221, SP124).
b) The (canconical) Herbrand universe is taken as (primary) semantics of sorts of specifications ('canonical term algebra'); see SEM_1 below.

We are now ready to state the semantics of an ASPIK specification.

Sem_1

M⟦s: Spec⟧$\xi\sigma$:=
(1) let s_id := (spec_id (sp_head s)),
$(u_1, \ldots, u_n) :=$ (use (sp_head s)),
$(s_1, \ldots, s_a) :=$ (sorts (sp_head s)),
$(o_1, \ldots, o_b) :=$ (ops (sp_head s)),
$(a_1, \ldots, a_c) :=$ (aux (sp_body s)),
$(c_1, \ldots, c_d) :=$ (cons (sp_body s)),
$(p_1, \ldots, p_e) :=$ (priv (sp_body s)),
$(aud_1, \ldots, aud_c) :=$ (def_aux (sp_body s)),
$(cad_1, \ldots, cad_a) :=$ (def_car (sp_body s)),
$(cod_1, \ldots, cod_d) :=$ (def_cons (sp_body s)),
$(opd_1, \ldots, opd_k) :=$ (def_ops (sp_body s)) in

(2) let $(\xi_1, \sigma_1) :=$ EXTEND(s)$\xi\sigma$ in

(3) let $U_i :=$ E⟦u_i⟧$\xi\sigma$, $i \in (n)$ in
let $U := \cup U_i$, $i \in (n)$ in

```
(4)  let (ξ0, σ0) := (ξ1, σ1), loci := newloc(ξi-1),
       ξi(si) := (loci, SORT, ⊥), σi := σi-1[loci ↤ ⊥],
                                        i ∈ (a)   in
     let (ξ0, σ0) := (ξa, σa), loci := newloc(ξi-1),
        ξi((op_id cadi)) := (loci, FUNC, ⊥),
        σi := σi-1[loci ↤ ⊥], i ∈ (a)  in
     let (ξ0, σ0) := (ξa, σa), loci := newloc(ξi-1),
        ξi((op_id oi)) := (loci, FUNC, ⊥),
        σi := σi-1[loci ↤ ⊥], i ∈ (b)  in
     let (ξ0, σ0) := (ξb, σb), loci := newloc(ξi-1),
        ξi((op_id ai)) := (loci, FUNC, ⊥),
        σi := σi-1[loci ↤ ⊥], i ∈ (c)  in
     let (ξ0, σ0) := (ξc, σc), loci := newloc(ξi-1),
        ξi((op_id pi)) := (loci, FUNC, ⊥),
        σi := σi-1[loci ↤ ⊥], i ∈ (e)  in
     let (ξ0, σ0) := (ξe, σe)  in
(5)  let (H1, ..., Ha) := H(s_id)ξ0σ0  in
     let (ξ1, σ1) := (ξ0, σ0[ξ0(s1)↓1 ↤ H1, ...
                              ξ0(sa)↓1 ↤ Ha]  in
(6)  let (A1, ..., Ac) := fix f1, ..., fc . λξ̄σ .
          let oidi := (op_id audi), i ∈ (c) in
            (E⟦(op_body aud1)⟧ξ̄σ[ξ̄(oid1)↓1 ↤ f1],
             . . .
            (E⟦(op_body audc)⟧ξ̄σ[ξ̄(oidc)↓1 ↤ fc]) in
     let (ξ1, σ1) := (ξ0, σ0[ξ0((op_id aud1))↓1 ↤ A1, ...,
                             ξ0((op_id audc))↓1 ↤ Ac]) in
(7)  let (C1, ..., Ca) := fix f1, ..., fa . λξ̄σ .
          let cidi := (op_id cadi), i ∈ (a)
               such that cidi corresponds to si in
            (E⟦(op_body cad1)⟧ξ̄σ[ξ̄(cid1)↓1 ↤ f1],
             . . .
            (E⟦(op_body cada)⟧ξ̄σ[ξ̄(cida)↓1 ↤ fa]) in
     let (ξ2, σ2) := (ξ1, σ1[ξ1((op_id cad1))↓1 ↤ C1, ...,
                             ξ1((op_id cada))↓1 ↤ Ca],
                             ξ1(s1)↓1 ↤ (|C1| ∪ {⊥s(1)}), ...,
                             ξ1(sa)↓1 ↤ (|Ca| ∪ {⊥s(a)}) in
(8)  let (Co1, ..., Cod) := fix f1, ..., fd . λξ̄σ .
          let oidi := (op_id codi), i ∈ (d) in
            (E⟦(op_body cod1)⟧ξ̄σ[ξ̄(oid1)↓1 ↤ f1],
             . . .
            (E⟦(op_body codd)⟧ξ̄σ[ξ̄(oidd)↓1 ↤ fd]) in
     let (ξ3, σ3) := (ξ2, σ2[ξ2((op_id cod1))↓1 ↤ Co1, ...
                             ξ2((op_id codd))↓1 ↤ Cod]) in
(9)  let (O1, ..., Ok) := fix f1, ..., fk . λξ̄σ .
          let oidi := (op_id opdi), i ∈ (k) in
            (E⟦(op_body opd1)⟧ξ̄σ[ξ̄(oid1)↓1 ↤ f1],
             . . .
            (E⟦(op_body opdk)⟧ξ̄σ[ξ̄(oidk)↓1 ↤ fk]) in
     let (ξ4, σ4) := (ξ3, σ3[ξ3((op_id opd1))↓1 ↤ O1, ...,
                             ξ3((op_id opdk))↓1 ↤ Ok]) in
(10) let C := {σ4(ξ4(s1)↓1), ..., σ4(ξ4(sa)↓1)}
       F := {A1, ..., Ac, C1, ..., Ca, Co1, ..., Cod, ...
             O1, ..., Ok}      in
     let loc := newloc(ξ4),
       ξ5 := ξ4[sp_id ↤ (loc, SPEC, ⊥)],
       σ5 := σ4[loc ↤ (C, F) ∪ U]  in
```

(ξ_S, σ_S)

Remarks: a) No context-sensitive correctness conditions are considered (see sec. 3.2.). Also type checking and scoping are disregarded.

b) The semantics of a specification is constructed as follows:

(1) Identifiers for important components are introduced by abstract syntax selections.

(2) Characteristic predefined (operation) identifiers of a cta-environment are supplied with syntactical information. This will be used in (5) where the operator H is applied to generate Herbrand universes for the new sorts.

(3) The semantic algebra derived from all used objects is generated. The case of used specterm objects is also covered although later only specterm-free specifications will be considered (see sec. 4.1.). The algebra U is well-defined, since in the case of constructive hierarchies every used object uniquely corresponds to a strict algebra $A \in Alg$, and the union of algebras then consists of set union (of carrier and operation sets). Since all identifers are globally unique (i.e. in ξ), unwanted identifications of carrier sets by the union process can be inhibited by an appropriate tagging of elements with the carrier set identifier. (For strict algebras and algebra union, see also [Olt 84b], sec. 2.2.1.).

(4) As a result of the ASPIK-specific separation of sort/operation identifier introduction (specification header) and sort/operation definition (specification body), at first slots are established in the environment that contain minimal information about each identifier.

(5) The Herbrand universe associated to each newly introduced sort is generated by application of the operator H (see definition 3.4.1.-5) and assigned as preliminary meaning to the sort identifier.

(6) The semantics of the auxiliary operations is generated by parallel fixpoint computation. Every operation body is functional (no state change s) such that E is applicable. The state $(\bar{\xi},\bar{\sigma})$ is assumed to contain information about actual parameter calling and passing (ASPIK parameters are called and passed by value). The resulting monotonous strict functions are bound to the auxiliary operation identifiers.

(7) Analogously to (6), but for the carrier predicates. Also, their extension - a restriction of the Herbrand universe - is bound to

associated sort identifiers.

(8) Analogously to (6), but for constructor operations.

(9) Analogously to (6), but for all remaining operations (= non-auxiliary, non-carrier predicates, non-constructors).

(10) Since E maps operation bodies to strict functions, all operations and sorts can be tied together in a strict algebra that also contains U. This object is assigned as semantics to the specification identifier.

The evaluation of operation bodies by E is defined in Sem_2.
We assume:
Let s ∈ Spec.
Let op denote an operation of s (public, private, auxiliary or carrier) with op := μ_0(op_id: Id, params: IdL, op_body: Op_body).
Let (ς, σ) ∈ State such that necessary context information for op is available (i.e. positions (6), (7), (8), (9) in Sem_1).
Let opb := (op_body op).

Sem_2: operation bodies

```
E⟦opb⟧ςσ :=
case opb ∈ Term : if (termL opb) = ⊥ then σ(ς(op_id opb)↓1)
                  else let (t1, ..., tn) := (termL opb) in
                   σ(ς(op_id opb)↓1)(E⟦t1⟧ςσ, ..., E⟦tn⟧ςσ)
case opb ∈ Case : let cv := (case_var opb),
                      (c1, ..., cn) := (caseL opb),
                      ti := (tag ci), exi := (exit ci),
                                              i ∈ (n) in
                   let cval := σ(ς(cv)↓1) in
                    if (∃̂ i ∈ (n) . ti matches cval)
                     then E⟦exi'⟧ςσ
                            where exi' is exi with
                             substitutions introduced by
                             the matching
                       else if (∃̂ i ∈ (n) .
                                   ti = 'OTHERWISE')
                            then E⟦exi⟧ςσ
                            else ⊥
case opb ∈ Cond : if E⟦(if opb)⟧ςσ
                  then E⟦(then opb)⟧ςσ
                  else E⟦(else opb)⟧ςσ
case opb ∈ Let : let lid := (let_var opb),
                     lt := (let_term opb),
                     bdy := (let_body opb)   in
                  let loc := newLoc(ς),
                      ς1 := ς[lid ↤ (loc, VAR, ⊥)],
                      σ1 := σ[loc ↤ E⟦lt⟧ςσ]  in
                E⟦bdy⟧ς1σ1
```

Remarks: a) The matching process occurring in the second case is the usual matching resulting in ground terms (note, that carriers consists of Herbrand universes, and variable values are terms of these carriers).
b) The refinement of Term1 is omitted.

By this definition the semantics of a specification is computed as a unique algebra. It is sometimes called the canonical term algebra.

Concerning the semantics of the remaining ASPIK objects (map-objects, imp-objects, and spec_terms) we proceed as follows: we skip imp-objects because they are not relevant for this paper and currently under research. Spec_terms in the general case possess a complex semantics that includes a 'normal-form-computation' and implicit object generations. They represent the parameterization concept of ASPIK, in which specifications with 'parameters' can be actualized (i.e. object parameterization). But it should be noted that no new kind of specification is generated by specterms: if all spec-objects involved in a specterm are describable by Sem_1 then the semantics of a specterm is also a canonical term algebra (hierarchy).

According to the intention of this paper it would be necessary for completeness to include the parameterization concepts of ASPIK and ModPascal in the treatment of a connection of applicative and procedural languages. But from the last paragraph it follows that the parameterization case can be reduced to the situation of specifications and their connection to modules. In fact, we will later (sec. 4.1.) restrict the class of spec-objects to specterm-free specs. It is clear that this diminishes the expressivity of the language and makes our concept less general, and actually we consider the treatment of parameterization only as postponed; the next iteration to this topic will include it. But for the moment we are freed from many technical burdens, and therefore we skip explicit semantic definitions of specterms and also of maps (although the latter do not cause problems).

3.4.2. ModPascal

The semantic clauses for ModPascal objects rely heavily upon the semantics of operation and type declarations. Here we give only the meanings of the objects introduced in sec. 3.1.2. (module type definitions, enrichment definitions, instantiation definitions and instantiate type definitions). Notions, operators, domains, variables, etc. that are not defined here can be found in [Olt 84b]. In the sequel the full semantics of ModPascal is assumed.

In addition to the semantics of [Olt 84b], syntactical operators are introduced that store information about the syntactic object. These data is used in the semantic clauses of section 4; therefore we cannot proceed in the usual way con-

sisting of generating a meaning from a syntactic object, and then forgetting all details. To deal with this issues, we introduce the notion of a mod-environment in which slots for syntactical operators exist that evaluate to the desired information if applied. Note that mod-environments for ModPascal are the analogon of cta-environments for ASPIK (see definition 3.4.1.-1). See also enr-environments, definition 3.4.2.-2.

3.4.2.-1 Def. [mod-environment]
Let MClauseOps ⊆ Id with
MClauseOps := {malg?, muse?, p_proc_id?, p_func_id?, p_init_id?, l_type_id?, l_proc_id?, l_func_id?, l_var_id?, p_proc_ar?, p_func_ar?, p_init_ar?, l_func_ar?, l_proc_ar?, l_var_type?, mop_def?, toi?}
Then ϱ ∈ Env is called mod-environment if for all x ∈ MClauseOps
a) ϱ(x) ≠ ⊥
b) ϱ(x)↓2 = MODSEL ¤

Remark: Associated to every element el of MClauseOps there is an ambiguously denoted special function el that evaluates to syntactical information if applied to module and module operation identifiers:

el = malg?
Associated operation: malg?: Id ⟶ State ⟶ D_BOOL
ϱ(malg?) = (loc, MODSEL, ⊥)
σ(loc) = {(id, tv)| id ∈ Id, tv ∈ D_BOOL}
malg?(id)ϱσ := σ(ϱ(malg?)↓1)(id)

el ∈ {muse?, p_proc_id, p_func_id?, p_init_id?, l_func_id?, l_init_id?, l_var_id?, l_type_id?}
Associated operation: el: Id ⟶ State ⟶ IdL
ϱ(el) = (loc, MODSEL, ⊥)
σ(loc) = {(id, (id$_1$, ..., id$_n$))| id, id$_i$ ∈ Id, i ∈ (n), n ∈ ℕ}
el(id)ϱσ := σ(ϱ(el)↓1)(id)

el ∈ {p_proc_ar?, p_func_ar?, p_init_ar?, l_func_ar?, l_proc_ar?}
Associated operation: el: Id ⟶ State ⟶ Arity
ϱ(el) = (loc, MODSEL, ⊥)
σ(loc) = {(id, ad)| id ∈ Id, ad ∈ ArDes}
el(id$_1$, id$_2$) := σ(ϱ(el)↓1)(id$_1$)(id$_2$)

el = l_var_type?
Associated operation: l_var_type?: Id ⟶ State ⟶ Id
ϱ(l_var_type?) = (loc, MODSEL, ⊥)
σ(loc) = {(id$_1$, id$_2$))| id$_i$ ∈ Id, i ∈ {1, 2}}
l_var_type?(id)ϱσ := σ(ϱ(l_var_type?)↓1)(id)

el = mop_def?
Associated operation: mop_def?: Id ⟶ State ⟶ OperationL
ϱ(mop_def?) = (loc, MODSEL, ⊥)
σ(loc) = {(id, (opd$_1$, ..., opd$_n$))| id ∈ Id, opd$_i$ ∈ Operation, i ∈ (n), n ∈ ℕ}

mop_def?(id)ξσ := σ(ξ(mop_def?)↓1)(id).

Note that all identifiers of modules, module operations and local variables are assumed to be unique.

¤

Based on mod-environments the operations EXT and EXTEND of section 3.4.1. are defined analogously except that EXTEND does not involve an updating of the toi? slot; since in the case of modules the type-of-interest is a semantical notion, toi? is of different (non-syntactical) quality and explicitly set in the semantic clause for module definitions (Sem_3 below).

A module type definition of a ModPascal program is embedded in a type definition scheme where a new type identifier is introduced to which the semantics of the definition is associated. We assume the identifier mid in Sem_3.

We are now ready to state the semantics of module type definitions:

Sem_3: Module type

```
MtⅡm: Module_type⅁ξσ :=
(1) let (u1, ..., ua) := (useL m),
        (p1, ..., pb) := (publicL m),
        (lt1, ..., ltc) := (local_typeL (local m)),
        (lv1, ..., lvd) := (local_varL (local m)),
        (lo1, ..., loe) := (local_operationL (local m)),
        (o1, ..., of) := (operationL m)
                                                          in
(2) let (ξ1, σ1) := EXTEND(m)ξσ                           in
(3) let U := ⋃ σ(ξ(ui)↓1)                                 in
            i∈(a)
(4) let (ξ0, σ0) := (ξ1, σ1)                              in

    let loci := newloc(ξi)
     where (case pi ∈ Proc_head :
                let opidi := (proc_id pi), obqi := PROC   in
            case pi ∈ Func_head :
                let opidi := (func_id pi), obqi := FUNC,
                    resi := (result pi)                   in
            case pi ∈ Init_head :
                let opidi := (init_id pi), obqi := INIT   in
             , i ∈ (b))
           ξi+1 := ξi[opidi ↤ (loci, obqi, if obqi = FUNC
                                           then resi
                                           else ⊥)],
           σi+1 := σ[loci ↤ ⊥], i ∈ (b)                   in
(5) let (ξ0, σ0) := (ξb, σb)                              in

    let loci := newloc(ξi), i ∈ (c)
     where ξi+1 := ξi[(typeid lti) ↤ (loci,
                                      (MtⅡ(type lti)⅁ξiσi)↓1,
                                     (MtⅡ(type lti)⅁ξiσi)↓2)],
```

```
            σ_{i+1} := σ_i[loc_i ↤ (Mt⟦(type lt_i)⟧ξ_i σ_i)↓3],
              i ∈ (c)                                                  in
(6) let (ξ_0, σ_0) := (ξ_c, σ_c)                                       in

    let (ξ_{i+1}, σ_{i+1}) := M⟦lv_i⟧ξ_i σ_i, i ∈ (d)                   in
    let LV := ⋃ (idL lv_i)                                             in
          i∈(d)
(7) let (ξ_0, σ_0) := (ξ_d, σ_d)                                       in

    let loc_i := newloc(ξ_i)                                           in
     where (case lo_i ∈ Proc_head :
                let opid_i := (proc_id lo_i), obq_i := PROC             in
           case lo_i ∈ Func_head :
                let opid_i := (func_id lo_i), obq_i := FUNC,
                    res_i := (result lo_i)                             in
            , i ∈ (e))
           ξ_{i+1} := ξ_i[obid_i ↤ (loc_i, obq_i, if obq_i = FUNC
                                                  then res_i
                                                  else ⊥)],
            σ_{i+1} := σ_i[loc_i ↤ ⊥], i ∈ (e)                         in
    let (ξ_0, σ_0) := (ξ_e, σ_e)                                       in

(8) (case o_i ∈ Proc_spec :
  let opid_i := (proc_id o_i), (pl_1, ..., pl_g):=(paramL o_i),
      D_i := (LV ∪ ⋃ (idL pl_j)) × LV                                  in
                j∈(g)
  case o_i ∈ Func_spec :
  let opid_i := (func_id o_i), (pl_1, ..., pl_g):=(paramL o_i),
      D_i := (LV ∪ ⋃ (idL pl_j)) × LV × ξ(opid_i)↓3                    in
                j∈(g)
  case o_i ∈ Init_spec :
  let opid_i := (init_id o_i), (pl_1, ..., pl_g):=(paramL o_i),
      D_i := (LV ∪ ⋃ (idL pl_j)) × LV                                  in
                j∈(g)
     , i ∈ (f))
(9) let (ST_1, ..., ST_f) := fix T_1, ..., T_f . λξ_1 σ_1 .
    (M⟦body o_1⟧ξ_1[opid_1 ↤ (ξ(opid_1)↓1, ξ(opid_1)↓2, ⊥)]
                σ_1[ξ(opid_1)↓1 ↤ ℛ({T_1, ..., T_f}, ξ_1, D_1)],
     ...
     M⟦body o_f⟧ξ_1[opid_f ↤ (ξ(opid_f)↓1, ξ(opid_f)↓2, ⊥)]
                σ_1[ξ(opid_f)↓1 ↤ ℛ({T_1, ..., T_f}, ξ_1, D_f)])
                                                                       in
 let opdef_i := ℛ(ST_i, ξ_i, D_i), i ∈ (f)                             in
 let (ξ_1, σ_1) := (ξ_0[opid_i ↤ (ξ(opid_i)↓1, ξ(opid_i)↓2, ⊥]
                   σ_0[σ_0(ξ_0(opid_i)↓1) ↤ opdef_i]),
                   i ∈ (f)                                             in

(10) let M-Val := X {ξ_1(id)↓3 | id ∈ LV}                              in
     let M-F := {σ_1(ξ_1(opid_i)↓1) | i ∈ (f)}                         in
     let M-Alg := ({M-Val}, M-F) ∪ ⋃ ∪ {ξ_1(typeid lt_i)↓1 |
                                          i ∈ (c)}                     in
    ((MODULE, M-Val, M-Alg), (ξ_1, σ_1))
```

Remarks: a) No context-sensitive conditions are considered (see sec. 3.2). Also type checking and scoping are

disregarded.

b) The semantics of a module type definition is constructed as follows:
   (1) Identifiers for important components are introduced by abstract syntax selections.
   (2) The syntactic information of m is embedded in the mod-environment; the module identifier mid is used.
   (3) The semantic algebra generated by the used objects is computed.
   (4) Locations for all public operation identifiers are reserved and supplied with initial values. The explicit binding of module operations in environments has only technical reasons (application of the fixpoint operator). It would suffice to install them directly as algebra functions.
   (5) The semantics of local types is computed and stored.
   (6) The local variable declarations are elaborated.
   (7) As (4) but for local operations.
   (8) The formal parameter lists of operations are computed; they will be used in (9) to denote the global, formal and result variables of an operation.
   (9) The semantics of all operations are computed by parallel fixpoint abstraction. By using the operator $\mathcal{R}$ the fixpoint is an algebra function defined on TOI's of local variable and parameter types. The state (ς_1, σ_1) is assumed to contain the appropriately called and passed formal parameter values.
   (10) The resulting algebra is built on the union of the used ones and equipped with the carrier generated from the cartesian product of the local variable TOI's and with all public and local operations.

c) Besides the module algebra, a resulting state is passed to save all parts of the definition. This makes convenient access possible in semantical clauses that are based on modules (e.g. enrichments, instantiations).

d) TOI(m) := M-Val; also in the embedding type definition with module identifier mid the type-of-interest is delivered to the toi?-slot of the mod-environment:
   by EXT(toi?, mid, M-Val)$\varsigma_1\sigma_1$, where $(\varsigma_1, \sigma_1) = (M[\![m]\!]\varsigma\sigma)\downarrow 2$.

We now present to the semantics of enrichments. Enrichments are also embedded in specific environments.

3.4.2.-2 Def. [enr-environment]
Let EClauseOps $\subseteq$ Id with
  EClauseOps := {enr?, euse?, add_id?, add_proc_id?,

add_func_id?, add_init_id?, add_proc_ar?, add_init_ar?, add_func_ar?, eop_def?}

Then $\xi \in$ Env is called enr-environment if for all $x \in$ EClauseOps

a) $\xi(x) \neq \perp$

b) $\xi(x)\downarrow 2 =$ ENRSEL ¤

Remark: Associated to every element el of EClauseOps is an ambiguously denoted special function el that evaluates to syntactical information if applied to enrichment and enrichment operation identifiers.

el = enr?
Associated operation: enr?: Id ⟶ State ⟶ D_BOOL
ξ(enr?) = (loc, ENRSEL, $\perp$)
σ(loc) = {(id, tv) | id $\in$ Id, tv $\in$ D_BOOL}
enr?(id)$\xi\sigma$:= $\sigma(\xi(\text{enr?})\downarrow 1)$(id)

el $\in$ {euse?, add_id?, add_proc_id, add_func_id?, add_init_id?}
Associated operation: el: Id ⟶ State ⟶ IdL
ξ(el) = (loc, ENRSEL, $\perp$)
σ(loc) = {(id, (id_1, ..., id_n)) | id, $id_i \in$ Id, i $\in$ (n), n $\in \mathbb{N}$}
el(id)$\xi\sigma$:= $\sigma(\xi(\text{el})\downarrow 1)$(id)

el $\in$ {add_proc_ar?, add_func_ar?, add_init_ar?}
Associated operation: el: Id ⟶ State ⟶ Arity
ξ(el) = (loc, ENRSEL, $\perp$)
σ(loc) = {(id, ad) | id $\in$ Id, ad $\in$ ArDes}
el(id_1, id_2) := $\sigma(\xi(\text{el})\downarrow 1)(id_1)(id_2)$

el = eop_def?
Associated operation: eop_def?: Id ⟶ State ⟶ OpDesL
ξ(eop_def?) = (loc, ENRSEL, $\perp$)
σ(loc) = {(id, (opd_1, ..., opd_n)) | id $\in$ Id, $opd_i \in$ OpDes, i $\in$ (n), n $\in \mathbb{N}$}
eop_def?(id)$\xi\sigma$:= $\sigma(\xi(\text{eop\_def?})\downarrow 1)$(id)

Note that all identifiers are assumed to be unique. ¤

Based on enr-environments the operators EXT and EXTEND of section 3.4.1. are defined analogously.

In Sem_5 the syntactical operator
AO: Public x Enrich_def ⟶ Id
is used. AO maps a public operation header p $\in$ (publicL a), a $\in$ (addL e), e $\in$ Enrich_def to that object identifier that is enlarged by the occurrence of p in its associated addpart of e:

AO(p, e) := ι id $\in$ Id .
let {a_1, ..., a_n} := (addL e) in
$\exists$ i $\in$ (n) . id = (add_id a_i) and
p $\in$ (publicL a_i)

The next clause introduces enrichments.

Sem_5: Enrichment definition

```
M⟦e: Enrich_def⟧ξσ :=
(1) let eid := (enr_id e), (u1, ..., ua) := (useL e),
      (a1, ..., ab) := (addL e),
      (o1, ..., oc) := (operationL e)  in
    let aidi := (add_id ai), i ∈ (b)   in
    let (pi1, ..., pib(i)) := (publicL ai), i ∈ (b) in
(2) let (ξ1, σ1) := EXTEND(e)ξσ   in
(3) let (ξ00, σ00) := (ξ1, σ1)                    in
    let locij := newloc(ξij)
       where
         (case pij ∈ Proc_head :
             let opidij := (proc_id pij), obqij := PROC in
          case pij ∈ Func_head :
             let opidij := (func_id pij), obqij := FUNC,
               resij := (result pij)                          in
          case pij ∈ Init_head :
             let opidij := (init_id pij), obqij := INIT in
         , i ∈ (b), j ∈ (bi))
         ξi,j+1 := ξij[opidij ↤ (locij, obqij,
                                 if obqij = FUNC
                                 then resij else ⊥)]
         σi,j+1 := σ[locij ↤ ⊥], i ∈ (b), j ∈ (bi) in
    if not AO(opidij, e) = aidi, i ∈ (b), j ∈ (bi) then ⊥
    else

(4) let (ξ0, σ0) := (ξbb(b), σbb(b))   in
    let (ξi+1, σi+1) := M⟦(paramL oi)⟧ξiσi, i ∈ (c) in

(5) let (ξ0, σ0) := (ξc, σc)   in
    (case oi ∈ Proc_spec :
     let opidi := (proc_id oi),
         (pl1, ..., plg) := (paramL oi),
         LVi := local variables of AD(opidi),
         Di := (LVi ∪ ⋃ (idL plj)) × LVi          in
                  j∈(g)
     case oi ∈ Func_spec :
     let opidi := (func_id oi),
         (pl1, ..., plg) := (paramL oi),
         LVi := local variables of AD(opidi),
         Di := (LVi ∪ ⋃ (idL plj)) × LVi × ξ0(opidi)↓3 in
                  j∈(g)
     case oi ∈ Init_spec :
     let opidi := (init_id oi),
         (pl1, ..., plg) := (paramL oi),
         LVi := local variables of AD(opidi),
         Di := (LVi ∪ ⋃ (idL plj)) × LVi          in
                  j∈(g)
   , i ∈ (c))

(6) let (ST1, ..., STc) := fix T1, ..., Tc . λξ1σ1 .
    (M⟦body o1⟧ξ1[opid1 ↤ (ξ(opid1)↓1, ξ(opid1)↓2, ⊥)]
                 σ1[ξ(opid1)↓1 ↤ ℛ({T1, ..., Tc}, ξ1, D1)],
      ...
```

```
      M⟦body o_c⟧⟦ς_1 [opid_c ↤ (ς(opid_c)↓1, ς(opid_c)↓2, ⊥)]
                  σ_1 [ς(opid_c)↓1 ↤ ℛ({T_1, ..., T_c}, ς_1, D_c)]⟧
                                                                  in
    let opdef_i := ℛ(ST_i, ς_i, D_i), i ∈ (c)                     in
    let (ς_1, σ_1) := (ς_0 [opid_i ↤ (ς(opid_i)↓1, ς(opid_i)↓2, ⊥]
                        σ_0 [σ_0(ς_0(opid_i)↓1) ↤ opdef_i]),
                        i ∈ (c)                                   in
(7) let U := ⋃ σ_1(ς_1(u_i)↓1)                                    in
             i∈(a)
(8) let E-F := {σ_1(ς_1(opid_i)↓1) | i ∈ (c)}                     in

    let loc := newloc(ς_1)                                        in
    let A := U ∪ (∅, E-F)                                         in
    let ς_2 := ς_1 [eid ↤ (loc, ENRICHMENT, ⊥)]                   in
        σ_2 := σ_1 [loc ↤ A,
                    ς(main)↓1 ↤ σ(ς(main)↓1) ∪ A]                 in

    (ς_2, σ_2)
```

Remarks: a) No context-sensitive conditions are considered (see sec. 3.2.). Also type checking and scoping are disregarded. The semantics exclude the case of enrichments of standard types with initial operations (see also [Olt 84a]).

b) The semantics of an enrichment definition is constructed as follows:
   (1) Identifiers for important components are introduced by abstract syntax selections.
   (2) The syntactic information of e is embedded in the enr-environment.
   (3) Locations for all introduced operation identifiers are reserved and supplied with initial values.
   (4) All parameter lists are evaluated.
   (5) The formal parameter lists of operations are computed; they will be used in (6) to denote the global, formal and result variables of an operation.
   (6) The semantics of all operations are computed by parallel fixpoint abstraction. By using the operator ℛ the fixpoint is an algebra function defined on TOI's of local variable and parameter types. The state (ς_1, σ_1) is assumed to contain the appropriately called and passed formal parameter values.
   (7) The semantic algebra generated by all used objects is computed.
   (8) The installation of the new object in the resulting state and the updating of the main program algebra (see [Olt 84b]) is done explicitly.

c) Enrichments do not possess a type-of-interest, since they are enlargements of several objects with several types-of-interest. Therefore the $\varsigma(eid)\downarrow 3$ component is assigned to $\bot$.

Since instantiations and instantiate types represent the ModPascal object parameterization concept, a similar remark as that following Sem_1 is applicable: problems occurring in the connection of parameterized structures are for the most part reducable to problems of the unparameterized case. Therefore parameterization is skipped here, but the treatment is only postponed. For sake of completeness we give semantical clauses of ModPascal instantiation objects and instantiate type definitions although they will be disregarded in section 4.

Instantiation definitions are defined as:

Sem_5: Instantiation Definition

```
M⟦i: Inst_def⟧ξσ :=
  let in_id := (inst_id i), (I1, ..., Ia) := (useL i),
      (ob1, ..., obb) := (ob_actL i),
      (t1, ..., tc) := (type_actL i),
      (op1, ..., opd) := (op_actL i)        in
  let (f, g) := σ(ξ(I1)↓1) + ... + σ(ξ(Ia)↓)1) in
   if not (SM?((f, g))) then ⊥
   else
    let F := {((old oi), (new oi))| i ∈ (b)} ∪
             {((old ti), (new ti))| i ∈ (d)}  in
    let G := {((old opi), (new opi))| i ∈ (c)} in
     if not (SM?((F, G))) then ⊥ else
      let SM := (f, g) + (F , G)  in
       if not (SM?(SM)) then ⊥ else
        let loc := newloc(ξ) in
        let ξ1  := ξ[in_id ↤ (loc, INST, ⊥, ⊥)]
            σ1  := σ[loc ↤ SM]   in
        let σ2  := σ1[σ1(ξ1(main)↓1) ↤ σ1(ξ1(main)↓1) ∪
                      ({source(SM), target(SM)}, {SM})] in
        (ξ1, σ2)
```

Remarks: a) SM? is the predicate to indicate signature morphism property of its argument (see [Olt 84b]).
c) For consistency and for verification contexts, an algrebra of the form above (last let-scheme) is added to 'main' (see [Olt 84b]).

The next clause introduces instantiate type definitions:

Sem_6: Instantiate Type Definition

```
Mm⟦i: Instantiate_type⟧ξσ :=
  let bid := (base_type i),
      {i1, ..., in} := (objectL i)   in
  let Bid := (Retrieve(bid)ξσ)↓1  in
  let {I1, ..., In} := RetOb({i1, ..., in})ξσ  in
  let I := I1 + ... + In,  I = (f, g)  in
    if not (SM?(I)) = true then ⊥ else
    if not (Comp?(Bid, I)) then ⊥ else
```

```
Let Bid1 := MARK(Ū(R̄u(Bid)), f) in
Let Bid2 := GENERATE(Bid1, g),
     {ob1, ..., obm} = Bid2 in
Let objL := SEQ({ob1, ..., obm}) in
Let (ς1, σ1) := M⟦objL⟧ς σ in
Let (A, (ς2, σ2)) := M⟦TOP(Bid2)⟧ς1 σ1 in
    (A, (ς2, σ2))
```

Remarks: a) The semantics of the base type and the used instantiation objects (both are elements of Id) are computed from the application state. By means of the Retrieve operator the associated syntactic objects are taken to perform the instantiation process (marking, object generation). The resulting object set is sequentialized and mapped to the appropriate semantic domain. The resulting state and the algebra of the TOP-element are passed (for the definition of operators, see [Olt 84b]).

b) All implicitly generated objects are installed. An appropriate naming procedure is assumed.

4. Connection and Correctness

We are now going to formulize the situation that was informally described in section 1: we assume a SEE with level languages ASPIK and ModPascal, and a stepwise-refinement methodology that at last connects specification objects with module objects. The connection will be defined by specific objects, the representation objects. We assume that all this information (specification, module, representation) is given (= supplied by the programmer), and then the central issue is to show a homomorphy condition that serves as notion of correctness for this refinement.

Section 4.1. introduces basic notions and confinements. Then foundations of abstract data types (with main emphasis on homomorphisms and algebras) are briefly reviewed in section 4.2. The syntactic and semantical definitions of representation objects are given in section 4.3., and realization conditions are introduced as sufficient conditions for correctness in 4.4. We close this section with an overview on other approaches to object correctness.

4.1. Confinements and Basic Notions

The need for the introduction of confinements into our approach arises from a substantial and a theoretical fact:

substantial: if constructs of applicative and procedural languages are going to be connected semantically, then (at least) the connection problems CP1 to CP3 of sec. 2.2. occur. But below this level, also care has to be taken to respect elementary characteristics of the language types (e.g. typing of expressions or scoping of variables). Provi-

sions must be made to limit side-effects implied by these characteristics.

technical: if we consider both languages with their full expressibility many boring issues have to be treated detailed although they do not contribute to a better insight in the approach (the topics range from variable renamings to, for example, rules for connecting call-by-values functions with call-by-reference functions). For clearness of presentation of this paper the attention is focussed on central em_6: Instantiate Type Definition | | points; doubtless, a future exhaustive description has to cover the languages completely.

To formulate our confinements we introduce some terms and phrases informally; the precise version follows in sec. 4.4.

4.1.-1 Terms/Phrases

(a) By external indication (programmer, user of a SEE) two objects of ASPIK and ModPascal may be characterized as being 'involved in a (b) An implementation is a refinement relation between two ASPIK spec objects that satisfies some conditions.

(c) A realization is a refinement relation between an ASPIK spec object and a ModPascal module object that satisfies some conditions. The term 'realization context' is used, if the validity of the conditions is uncertain. ¤

With this terminology we are able to state our restrictions:

R1: Every spec-object involved in a realization context is algorithmic

For the justification of R1 we have to go into deeper details of the ASPIK semantics.

The classification of specs into axiomatic and algorithmic ones is not only a syntactical question (presence/absence of the specification body). It is used to assign different semantical structures such that axiomatic specs may be characterized as 'more general' than algorithmic specs. That term requires precision.

Let an algebraic specification SP be a triple (S, O, C) consisting of a set S of sort names, a set O of operation names with operation functionalities, and a set C of constraints built from the symbols of S, O, and specific predefined symbols. The tuple (S, O) is also called signature (see sec. 4.2.). It is easy to see that an ASPIK spec is an algebraic specification in this sense: the sets S and O are given by the sorts- and ops-clause of the spec hierarchy, and the constraints are either the definitions of the spec body or contained in the props-clause (since ASPIK allows predicate

calculus as property language, predefined symbols of constraints are, for example, the logical connectives and qualifiers).

In the initial approach, the set C of constraints is usually a set of equations (or at most universal horn clauses); ASPIK is more powerful in this point. Now, independently from the specific approach algebras are considered with respect to a given signature SIG = (S, O), i.e. algebras that possess carrier sets associated to the names of S, and operations associated to the names (and functionalities) of O. The set of these algebras is the object set of a category Alg[SIG]. An important subcategory of Alg[SIG] is defined by those algebras A $\in$ Alg[SIG] that satisfy the constraints of C. We denote it by Alg[SIG, C].

At this point differences among approaches to the semantics of algebraic specifications become significant: If an axiomatic spec S induces a signature SIG and constraints C, then the initial algebra approach would select as semantics of S the (isomorphism class of the) initial object in Alg[SIG, C]. The initial object is a unique element of that category, and therefore the semantics of the specification is rather fixed. Instead of, ASPIK employs the 'loose' approach. In order to avoid a characteristic of the initial algebra approach that forces the programmers attention to one determined object at the very first steps of software development (as represented by axiomatic specs) - a fact which in our view essentialy diminishes the benefits of abstraction -, the ASPIK semantics chooses the <u>whole</u> class Alg[SIG, C] as semantics of S. Then an axiomatic spec is also described by non-initial algebras; every element of Alg[SIG, C] can be taken as meaning of S in arbitrary applications.

The situation changes, if algorithmic specs are considered. Since they are intended to describe a specific algebra more concrete, the ASPIK semantics constructs a canonical term algebra (CTA). Its carriers consist of (subsets of) the Herbrand universes generated from the elements of the constructor clause. Its operations are derived from all operation definitions of the spec, such that violations of closedness conditions do not take place. Then Alg[SIG, C] consists only of the isomorphism class of the CTA; then a trivial consequence is that canonical term algebras are initial objects in their categories. In some sense, algorithmic specs represent a final state in the development and refinement process: their semantics is (up to isomorphism) a specific object and there are no other models available. (If nevertheless algorithmic specs are refined into other algorithmic specs, the term 'implementation' and the ASPIK implementation concept are applied).

This brings the interlude on the ASPIK semantics to its end; a detailed presentation of the material can be found in [BV 85].

How is R1 justified by that definitions? If we allow axiomatic

specs occurring in realization contexts, we have to

- develop a concept of realization of a set of algebras (spec semantics) by a single algebra (module/enrichment semantics), or to
- select one possible algebra.

The first alternative is more general, and it has inheritant deeper theoretical issues as well as clearness decreasing complexity. But the main counter argument is that in SEEs realization steps of this 'size' (high abstraction to concrete representation) are unrealistic; software development processes are more continuous, and all efforts in software engineering try to avoid transparency reducing refinements of this kind.

But selection of a specific algebra comes with other problems: what criterion has to be applied and how good is it? We assume that this question can be answered satisfactorily, and that the initial algebra of the appropriate category is choosen (this criterion is acceptable since many concepts of abstract data type theory are based on initial algebras, and therefore results can be applied). If we are going to define the realization conditions, we have to have a concrete representation of the initial object. Since it is induced 'only' by an axiomatic spec there is in general no (constructive) way to generate such a representation, due to the undecidability of predicate calculus. In the special case that only equations are used as properties of specification, then spec the only representation we have is the so-called 'quotient term algebra' (QTA): the carriers are congruence classes generated over the Herbrand universe by the constraint set of the spec, and the generations are defined on the congruences.

Then, if proof tasks have to be processed they would have to deal with congruence classes of terms, not with terms directly. This is highly unwanted, since then

- proof systems have to include the theory of congruences,
- every derivation has to be checked for independence of the representative, and
- the connection of the module algebra and the spec algebra becomes more unnatural by linking a single concrete representation (carrier element) to a set of abstract terms; usually, it is the other way around.

It is not clear to us if satisfactory solutions to this issues exist for SEEs.

In this light the advantages of restricting realizations to algorithmic specifications with CTAs as semantics have to be seen:

- carriers and operations of the semantical algebra are constructive: the former are given by the Herbrand universes (or restrictions thereof), the latter are just the algor-

ithmic operation definitions given by the programmer in the spec object definition;
- the module algebra carriers and the CTA carriers can be linked by associating single representations to single terms;
- the restrictions imposed on carrier sets of CTA allow to use structural induction as proof method; therefore
- proof tasks can be treated by existing systems without greater modifications. This includes that there is a choice between different proof methods implemented by different systems (theorem provers, induction provers, rewrite rule derivations), and one possesses a greater flexibility to adapt an appropriate method to a given task.
- it is very easy to 'execute' algorithmic specifications by interpreters, and by this there is strong support of a testing tool of a SEE.

The next restriction concerns the sort clause of an algorithmic specification.

R2: Every spec-object introduces at most one new sort

The reason for this restiction lies in a principle property of procedural, strongly typed languages.
Consider the case of module constructs of ModPascal. Unlike similar constructs for object oriented programming languages (ADA packages, Modula-2 modules), ModPascal modules are incarnatable: variables might be declared of a module type and used in statements and expressions. (Simula classes or CLU clusters are incarnatable; but both concepts do not fit as imperative counterpart to ASPIK). By the declaration, the association variable-type is fixed in the scope and extent of the variable. Semantically, this is modelled by the fact that the module algebra possesses one special carrier (the cartesian product carrier) which supplies module incarnations with values. Every module definition introduces just one of these special carriers, and thus may only be used for a refinement of specifications with one sort.

If a spec S contains (at least) two different sorts in its sort-clause, the CTA A(S) has (at least) two different carriers. In the specification language this causes no problems: there are no spec-variables (i.e. variables taking their values out of a set of spec names) but only sort variables. ASPIK terms are connected to a unique sort and not to a spec.

This is not true in the case of ModPascal (and any imperative language with incarnatable module construct): there is the notion of a module type, and variables are declared being thereof; since only one new data set is introduced by a module type definition, there is no choice for the value set of variables. (Admittedly, this design decision could be modified such that several cartesian product carriers are introduced by

a module type definition, but at what a price: either one would loose the classical feature of variables as object incarnations, or 'over-loading' of variables becomes possible. In general, it would not be distinguishable for a given construct from which cartesian product carrier a module variable takes its values. This is not a practicable road.)

Having these preconditions in mind it is obvious to impose restriction R2 on specifications occurring in realization contexts. Then there is also no difficulty in associating abstract and concrete data as demanded by the technique of algebraic verification (see secs. 1.3. and 4.4.): there is a 'real' one-to-one mapping between the single abstract and the single concrete carrier.

It should be emphasized that R2 represents a serious limitation of the approach: in SEEs, one has to impose an artificial structure on ones problem solution to meet this requirement, and possibly looses some of the benefits gained by the application of algebraic specifications. Also there is a (non-trivial) subclass of problems that become unspecifiable in ASPIK if introduction of more than one sort at a time is forbidden. We therefore try to find (for the subsequent iterations of the approach) a more satisfactory solution that treats incarnability and multiple carrier introduction; for the moment we require R2.

R3: Specterms do not occur

Specterms may occur in the use clause of ASPIK specs. Semantically, they denote new spec objects that are generated by application of the map component to the source spec. In this sense specifications with specterm occurrences are isomomorphic to specterm-free specs: if the structures behinds specterms are installed in the environment as (simple) spec objects, then these objects (more precise: object names) can be appropriately substituted for specterms yielding specterm-free specs.

This fact allows to remove the complexity induced by specterms in realization contexts. Issues such as normal form computation or implicit object generation need not be considered here, since equivalent and more simple objects are available. Furthermore, specterms are the constructs of the ASPIK parameterization concept for specifications, and the properties and correctness criteria of parameterization realization are not topic of this paper.

R4: Modules and enrichments do not contain instantiate type definitions

This is the analogon of R3 for ModPascal objects occurring in

realization contexts; the same arguments apply.

Remark: The renunciation of specterms implies the absence of map- (and imp-) objects; the renunciation of instantiate type definitions implies the absence of instantiation definitions.

We summarize our restrictions for objects occurring in realization contexts as a precision of definition 4.1.-1:

4.1.-2 Def. [realization, realization context]
Let S denote an ASPIK spec, M a ModPascal module or enrichment.
Let (S, M, true) denote an unspecified relation between S and M, where the boolean value 'true' indicates that no conditions need hold.
Let C_1 denote the conditions
R1: S is algorithmic
R2: S is single-sorted
R3: S is specterm-free
R4: M is instantiate_type_definition-free
Let C_2 denote (up to now) unspecified semantical conditions.

Then the triple (S, M, C_1) is called realization context (re-co) if S and M satisfy C_1.
The triple (S, M, C_1 and C_2) is called realization, if S and M satisfy both C_1 and C_2. ¤

Remark: The condition set C_2 will be made concrete in sec. 4.4. It will turn out to be the correctness criteria for the refinement of specifications into modules.

4.2. Homomorphisms and Algebras

To allow an exact formulation of our concept we now give definitions of basic notions of abstract data type theory. We are able to use them in our context since ASPIK as well as ModPascal features have been designed in close connection to this theory, and both employ an algebraic semantics.

4.2.-1 Def. [signature]
Let OB, OP denote sets of (object and operation) identifiers.
Let arity: OP —> (OB* x OB) be defined.
Then the triple (OB, OP, arity) is called signature. ¤

Remarks: a) Arity associates a functionality to an operation name. In general, the functionality may be multi-valued, i.e. arity: OP —> (OB* x OB*).
b) A signature introduces only names for items, not items with a specific meaning.

Notations: Let (OB, OP, arity) denote a signature.
$OP_{s,t} := \{op \mid op \in OP \text{ and } arity(op) = (s, t)\}$
For arity(op) = (s, t), s is called source, and t target of op.
$\varepsilon \in OB^*$ denotes the empty source.

In the following we assume arity as function (exclusion of overloading). Then it follows, that $OP = \cup\{OP_{s,t} \mid s \in OB^*, t \in OB\}$ and $\cap\{OP_{s,t} \mid s \in OB^*, t \in OB\} = \emptyset$.

4.2.-2 Def. [signature morphism]
Let $(OB_i, OP_i, arity_i)$, $i \in \{1, 2\}$ denote signatures.
Then, a pair $(f: OB_1 \longrightarrow OB_2, g: OP_1 \longrightarrow OP_2)$ of functions is called signature morphism if it holds:
for all $op \in OP_1$ with $arity_1(op) = (ob_1 \ldots ob_n, ob)$.
$arity_2(g(op)) = (f(ob_1) \ldots f(ob_n), f(ob))$ ¤

Signature morphisms are arity-preserving functions between signatures.

The next definitions are needed to associate signatures with an appropriate semantical domain.

4.2.-3 Def. [flat domain]
Let S denote a set. Then $(S_\perp, \sqsubseteq)$ is called a flat domain
if 1) $\perp_s \notin S$ denotes the bottom element of S.
$S_\perp := S \cup \{\perp_s\}$
2) $\sqsubseteq \subseteq (S_\perp \times S_\perp)$ is a partial order with
$X \sqsubseteq Y :\Longleftrightarrow X = \perp_s$ or $X = Y$ ¤

Notation: If no ambiguities are possible, we denote the flat domain $S_\perp$ simply as S and the bottom element $\perp_s$ as $\perp$.

4.2.-4 Def. [strict]
Let $C_1, \ldots, C_m$ denote flat domains, and $n \in (m)$. A function
$f: C_1 \times \ldots \times C_n \longrightarrow C_{n+1} \times \ldots \times C_m$
is called strict, if
$f(c_1, \ldots, c_n) = (\perp_{c_{n+1}}, \ldots, \perp_{c_m}) \Longleftrightarrow \exists\, i \in (n) \,.\, c_i = \perp_{c_i}$
¤

Remark: Multi-value functions are considered because sometimes module operations take them as semantics.

4.2.-5 Def. [monotonic]
Let $f: C_1 \times \ldots \times C_n \longrightarrow C_{n+1} \times \ldots \times C_m$ be a strict function.
Let $X_i, Y_i \in C_i$, $i \in (n)$.
Then f is called monotonic if
$X_i \sqsubseteq Y_i$, $i \in (n) \Rightarrow f(x_1, \ldots, x_n) \sqsubseteq f(y_1, \ldots, y_n)$ holds.
¤

4.2.-6 Def. [continuous]
Let $f: C_1 \times \ldots \times C_n \longrightarrow C_{n+1} \times \ldots \times C_m$ be a monotonic function.
Let $X_i \subseteq C_i$ such that for all a, $b \in X_i$ either $a \sqsubseteq b$ or $b \sqsubseteq a$ holds.
Then f is called continuous if
$f(\sqcup X_1, \ldots, \sqcup X_n) = \sqcup\{f(x_1, \ldots, x_n) \mid x_i \in X_i\}$
where $\sqcup$ denotes the least upper bound. ¤

Remark: Since we consider only flat domains the sets X_i contain at most two elements.

4.2.-8 Fact
Let C_i, $i \in \{1, 2\}$ denote flat domains, and
$f: C_1 \longrightarrow C_2$. Then it is equivalent:
- f is continuous
- f is strict or f is constant. ¤

This ensures the well-definedness of those functions f we will later use as meaning of operation definitions op: if the strictness of f can be guaranteed, the continuity implies the existence of a least fixed point which can be taken as unique meaning of op.

We now deviate from the usual way and introduce algebras not over given signatures. Instead we define them explicitly and derive an associated signatur e in a second step.
meaning.

The reasons for this modification are that ModPascal modules are associated constructively to algebras (without e.g. term-generated carriers) and that there is no way to characterize these algebras in their categories (since no equations/axioms are available for modules). We will use the associated signatures of a given algebra A only to 'forget' specific data and operations of A, With this prerequisites we can define the notion of strict algebras. They will be used as semantical objects assigned to specification objects or module type definitions.

4.2.-9 Def. [strict algebra]
Let $C = \{C_1, \ldots, C_n\}$ denote a non-empty set of flat domains C_i, $i \in (n)$.
Let $F = \{\langle f: C_{i(1)} \times \ldots \times C_{i(m)} \longrightarrow C_{i(m+1)}\rangle \mid C_{i(j)} \in C$ and $\{i_1, \ldots, i_{m+1}\} \subseteq \{1, \ldots, n\}\}$ denote a set of strict continuous functions.
Then the tuple (C, F) is called strict algebra. The elements of C are called carriersets or carriers.

¤

Strict algebras can be associated to a signature under the assumption that there exist naming functions.

4.2.-10 Def. [naming functions]
Let A = (C, F) denote an strict algebra, and Id an unbound set of identifiers.
Then the naming functions
obname-A: C ⟶ Id and opname-A: F ⟶ Id
associate unique names to carrier sets and operations of A.
obnames(A) := {obn | ∃ c ∈ C . obn = obname-A(c)}
opnames(A) := {opn | ∃ f ∈ F . opn = opname-A(f)}

¤

4.2.-11 Def. [associated signature]
Let A = (C, F) denote an strict algebra.
Let arity-A: opnames-A ⟶ (obnames-A* x obnames-A) be defined by:
(f: $C_1 \times \ldots \times C_m \longrightarrow C_{m+1}$) ∈ F and opname-A(f) = $\bar{f}$
⇒ arity-A($\bar{f}$) = ($C_1 \ldots C_m$, C_{m+1})

Then $\Sigma(A)$ = (obnames-A, opnames-A, arity-A) is called the associated signature to A w.r.t. (ob_name-A, op_name-A).

¤

Now we consider strict algebras and arbitrary signatures.

4.2.-12 Def. [Σ-algebra]
Let Σ denote a signature.
Let A denote an strict algebra.
Then A is called Σ-Algebra if there exists a signature morphism (f: $\Sigma\downarrow 1 \longrightarrow \Sigma(A)\downarrow 1$, g: $\Sigma\downarrow 2 \longrightarrow \Sigma(A)\downarrow 2$) such that f and g are bijective.

¤

4.2.-13 Def. [Alg[Σ], Alg]
Let Σ denote a signature.
Then

$$Alg[\Sigma] := \{A \mid A \text{ is } \Sigma\text{-algebra}\} \cup \{\perp_\Sigma\}$$
$$Alg := + \{Alg[\bar{\Sigma}] \mid \bar{\Sigma} \text{ is signature}\}$$

(+ denotes the coalesced sum of domains). ¤

The definition of the domain Alg as coalesced sum of Σ-sorted algebra domains is not unproblematic. It would allow algebras that possess as carriers "the set of all sets". Since this is a well-known paradoxon-generating construction, we assume a meta-structure called 'universum' U whose elements are sets. There are axioms that make the "set of all sets" underivable in U. Then, all carriers of elements $A \in Alg$ are assumed to be elements of U.

We now need a relation between algebras of Alg. Since associated signatures may differ we look at common subsignatures and subalgebras.

4.2.-14 Def. [subsignature]
Let Σ = (OB, OP, arity) denote a signature.
Then $s\Sigma$ = (OB_1, OP_1, $arity_1$) is called subsignature of Σ if $OB_1 \subseteq OB$, $OP_1 \subseteq OP$, $arity_1$: $OP_1 \longrightarrow (OB_1^* \times OB_1)$ such that $arity_1(op) = arity(op)$ for all $op \in OP_1$.

¤

Subsignatures are used to modify algebras: a given strict algebra is reduced to a subalgebra described by a subsignature; module algebras (the algebras associated to module type definitions) will be treated in this way.

4.2.-15 Def. [subalgebra]
Let Σ denote a signature, and $A \in Alg$.
Then the Σ-subalgebra of A is defined by
(1) A is Σ-algebra: A
(2) A is not Σ-algebra:
(2.1) There exists a subsignature $s\Sigma(A)$ of $\Sigma(A)$ such that a signature morphism (f: $\Sigma\downarrow 1 \longrightarrow s\Sigma(A)\downarrow 1$, g: $\Sigma\downarrow 2 \longrightarrow s\Sigma(A)\downarrow 2$) with f bijective:

(C, O) where

$C := \{ca \mid ca \in A\downarrow 1 \text{ and } obname\text{-}A(ca) \in s\Sigma(A)\downarrow 1\}$
$O := \{op \mid op \in A\downarrow 2 \text{ and } opname\text{-}A(op) \in s\Sigma(A)\downarrow 2\}$

(2.2) otherwise: $\perp_{Alg}$

¤

Remark: If A is a Σ-algebra with $\Sigma(A) = \Sigma$ (i.e. the bijective signature morphism is the identity), and if Σ_1 is subsignature of Σ, then there is always a Σ_1 subalgebra of A.

Notation: The symbol '$\subseteq$' is ambiguously used to describe the subsignature relation ($\Sigma_1 \subseteq \Sigma_2$) as well as the subalgebra relation ($A_1 \subseteq A_2$).

Note that several Σ-algebras of a given algebra A may exist (e.g. if operations of A have the same functionality or 'plenty' carriers occur).
$Sub(\Sigma, A)$ denotes the set of all Σ-subalgebras of A.

We now connect algebras by homomorphisms. Again, we deviate from the usual path and modify the notion of algebra homomorphism in two ways: we assume that source object and target object are standing in a subalgebra relation, and we define only partial mappings between the carriers.
The subalgebra assumption was implied by the application of algebra homomorphisms in representation objects (see sec. 4.3.); partiality excludes those elements of source object carriers that are mapped to bottom elements. Again, in the application lateron we will not consider these elements (because they do not contribute to the desired correctness proposition).

4.2.-16 Def. [partial algebra homomorphism]
Let $A_i \in Alg$, $i \in \{1, 2\}$.
Then a family H of mappings $\langle h_{c(1)}: C_1 \longrightarrow C_2 \mid C_1 \in A_1\downarrow 1, C_2 \in A_2\downarrow 1\rangle$ is called partial algebra homomorphism, if

(1) There is a $\Sigma(A_1)$ subalgebra of A_2, denoted by sA_2. (f, g): $\Sigma(A_1) \longrightarrow \Sigma(sA_2)$ denotes the bijective signature morphism.
(2) For all $(h_{c(1)}: C_1 \longrightarrow C_2) \in H$ it holds:
  (2.1) $C_2 \in (sA_2)\downarrow 1$
  (2.2) $obname\text{-}A_2(C_2) = f(obname\text{-}A_1(C_1))$
(3) For every $ca \in (A_1)\downarrow 1$ there is $(h_{c(1)}: C_1 \longrightarrow C_2)$ such that $ca = C_1$.
(4) For all $op \in (A_1)\downarrow 2$ with $arity\text{-}\Sigma(A_1)(opname\text{-}A_1(op)) = (C_1 \ldots C_m, C_{m+1})$ there exists by means of (f, g) $op' \in (sA_2)\downarrow 2$ with $arity\text{-}\Sigma(sA_2)(opname\text{-}sA_2(op')) = (f(obname\text{-}A_1(C_1)) \ldots f(obname\text{-}A_1(C_m)), f(obname\text{-}A_1(C_{m+1})))$ and $opname\text{-}sA_2(op') = g(opname\text{-}A_1(op))$.
  Then, for all $ce_i \in C_i$, with $h(ce_i) \neq \perp_a$, $a := obname\text{-}A_2(f(obname\text{-}A_1(C_i)))$, $i \in (m)$ and for all $op \in (A_1)\downarrow 2$
  $h_{c(m+1)}(op(ce_1, \ldots, ce_m)) = op'(h_{c(1)}(ce_1), \ldots, h_{c(m)}(ce_m))$

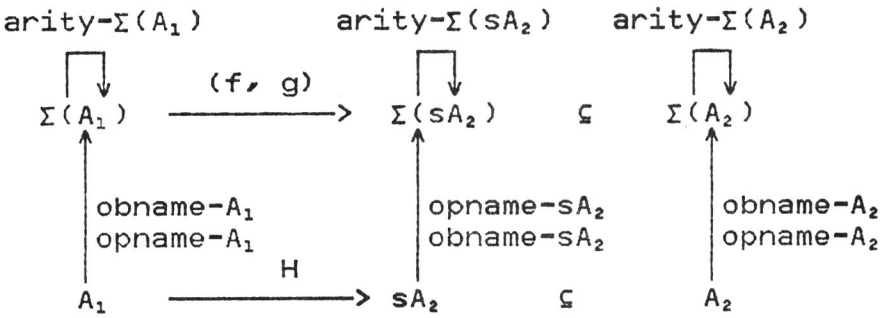
¤

Remarks: a) This definition is tailored to many-sorted algebras as they are contained in Alg. The one-sorted case is characterized by far less technical details.

b) It only makes sense to consider homomorphims between algebras with appropriate associated signatures, i.e. isomorphic signatures. Therefore A_2 has to be reduced to a subalgebra with this property and the homomorphism is only defined between sA_2 and A_1.
This reflects the case that the module object M occurring in a realization context (S, M, C) has a - loosely spoken - richer structure that S i.e. introduces more data and/or more operations. Then it is necessary to cut off the overlapping edges.

c) The homomorphism property is only considered on elements of C_i that are mapped to non-bottom elements by $h_{c(i)}$. This implies partiality of the homomorphisms, and it is used lateron to partition cartesian product carriers into elements that should correspond to an abstract carrier element, and those that should not (see sec. 4.3.).

d) The interaction between the various syntactical and semantical operators may be visualized as following:



As definition 4.2.-16 shows there is a close relation between algebra homomorphisms and signature morphisms. Especially the subalgebra-generating bijections are of great importance: if $\Sigma_1 \neq \Sigma_2$ and $Sub(\Sigma_1, A_2) \neq \emptyset$, then it depends on the choice of the bijection (and therefore $sA_2 \in Sub(\Sigma_1, A_2)$) if homomorphy can be shown with a given H. This fact may be used to strengthen the notion of algebra morphism by demanding property (4) for every element of $Sub(\Sigma_1, A_2)$. For our applications the weak version suffices.

In general the carriers of two algebras are not isomorphic. In the context of realizations this means that there is no information about a relation between the specification carriers and the module carrier. Especially the questions if the data introduced by the module is 'suffient' enough or is 'too large' (i.e. contains elements of no interest) cannot be answered.

In the case of algebra homomorphisms the situation looks better. Every homomorphism h: S -> T can be used to factorize its

source S to an isomorphic object S'. Then it can be shown that T is also isomorphic to S'. We formalize these ideas in sec. 4.4. when looking at realization conditions.

4.3. Representation Objects

The last section has introduced a number of concepts as signature morphism or algebra homomorphism that will be used as basis of our notion of realization. But up to now we have not said how the connection of ASPIK specifications and ModPascal modules/enrichments is given. This section is dedicated to this task: The concept of a representation object (rep-object) is presented in sec. 4.3.1., whereas the subsequent subsections treat abstract syntax, context-sensitive conditions and semantics resp. (secs. 4.3.2. to 4.3.4.).

4.3.1. Concept

A rep-object is a syntactic unit in which information about the connection between a spec and a module/enrichment is gathered. This information may be splitted into:

- information about the relation between operations of the spec and operations of the module/enrichment (= signature morphism)
- information about the relation between elements of spec carriers and elements of module carriers (= carrier mappings).

One could presume that with these ingredients the condition for algebra homomorphisms are directly satisfied (see definition 4.2.-16) but this is not true. Since we look at a very specific situation (refinement of specifications into modules/enrichments), rep-objects also include specific modifications.

What information should be given by a programmer who refines a spec S into a module M?

Firstly he should say for every operation of S which is the operation of M that is intended to refine it. Or with other words: we require the signature morphism sm going from (the signature of) S to (the signature) of M. If sm: M -> S, this would correspond to a refinement in reverse direction (i.e. an abstraction step). This is also an important scenario, but outside the scope of this paper.

Secondly he should say how values are connected. If he designs the refining structure (the module) he has to consider carrier elements of CTA(S) and carrier elements of Malg(M) (the module algebra associated to M; its carriers are cartesian products of those carriers that are associated to the types of the local variables of M; see [Olt 84b] for details). There are two possibilities for such a representation function (rep-

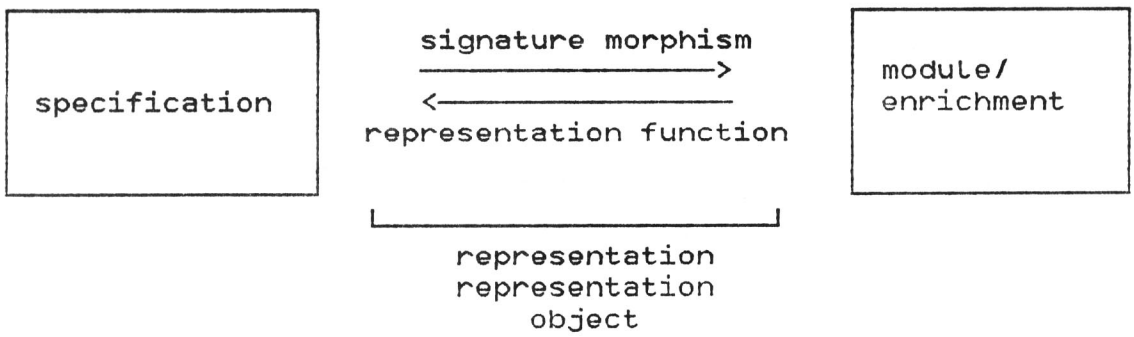
function) Rf between CTA(S) and Malg(M):

a) Rf: CTA(S) ——> Malg(M): This implies that abstract data can be uniquely represented by concrete data. If this is true, then refining is just renaming, and no sophisticated semantical criteria are necessary. But this is not the case in general.
Problematically also is the treatment of m ∈ Malg(m) with: for all s ∈ CTA(S) . Rf(s) ≠ m (i.e. the surjectivity of Rf). This data has no connection to abstract data at all, but this property is not visible and module operations may work well on such arguments. The only solution is to exclude them from the correctness check considerations, what is equivalent to moving from total to partial operations. But then also total functions of S are connected to partial operations of M, and this is not intended in a refinement (the expressability should not decrease).

b) Rf: Malg(M) -> CTA(S): Here, concrete data (vectors) is mapped to abstract values (terms). This way allows to represent a single term in different ways, or to explicitly disregard concrete information if intended. This is advantageous since refinements are often performed by modules/enrichments with more or less redundancy (since the cartesian product carrier is not further restricted, or predefined types of the language offer more operation and data types than needed). In these cases Rf can be used to tailor the carriers by identification of equivalent concrete data or by mapping redundant data to bottom elements of the abstract carriers.
The surjectivity of Rf assuring that <u>all</u> abstract data is in fact refined is also not guaranteed by this functionality of Rf (especially because Rf is assumed to be given by the 'unperfect' programmer). But under certain circumstances surjectivity is derivable in our approach (see sec 4.4.).

The surjectivity of Rf is a very important and necessary property in the SEE context. It ensures that no 'abstract' data is 'forgotten' i.e. has no 'concrete' counterpart. Otherwise it would be impossible to check the preservation of specified properties since the data that carries it is missing. We look at surjectivity of Rf (or an analogous function) as an essential requirement for a correctness notion.

In the following we assume Rf having functionality as in case b) above. Note that signature morphism and rep-function map into different directions:



If we want to employ rep-objects as algebra partial homomorphisms we have to slightly modify the above definition 4.2.-16 to meet the technical constraints (functionalities of signature morphisms and carrier mappings; see sec. 4.4.). But then rep-objects may be seen as the syntactic vehicle to define a family of mappings that is intended to describe a partial algebra homomorphism. If this is in fact the case remains to be shown, with the help of proving tools of the SEE for example. Depending on the outcome correctness of the refinement is achieved or not; we discuss this notion in sec. 4.4.

It should be noted that rep-objects may form hierarchies. Like specifications and modules/enrichments, they possess use-clauses in which other (rep-)objects may occur. The effect is, that all used signature morphisms and all used rep-functions are visible and have to be respected in a given rep-object. This in convenient because it allows to partition the rep-object design into substructures, and it is more natural since the involved specification and modules are hierarchical. Unfortunately, technical issues become more complex, because objects and hierarchy relations have to be considered.

The hierarchical structure of all kinds of objects is exploitet in the proof method of sec. 5.2. where a bottom-up procedure is proposed.

In other approaches to the correctness of object relations, rep-objects in this sense do not occur as independent objects. There, the necessary information is provided and gathered at several places, sometimes in the objects and sometimes in the method (see sec. 4.5.). We believe that this explicit presentation is best suited to the needs of SEEs and it emphasizes the importance of the rep-object information by assigning an own structure to it.

4.3.2. Abstract Syntax

A rep-object is described syntactically by the following abstract grammar:

```
Repobj          = (rob_id : Id, connect : Connect,
                   useL : IdL, operationL : OperationL,
                   rep_fct : Rep_fct)
Connect         = (source : Id, target : Id)
Operation       = (from : Id, to :Id)
Rep_fct         = (rf_id : Id, paramL : IdL,
                  rf_body : Rf_body)
Rf_body         = (let_schemeL : Let_schemeL,
                  a_term : A_term) v If
Let_scheme      = (let_var : Id, let_body : Let_body)
Let_body        = A_term v StmtL
A_term          = (at_id : Id, exprL : ExprL) v (a_id : Id,
                  a_termL : A_termL)
Expr            = Id v Term v S_term v Const_val
Term            = Simple_term v Op_designator
Simple_term     = (op_id : Op_id, paramL : ExprL)
Op_id           = Id v Pre_id
Op_designator   = (var_id : Id, op_idL : IdL, paramL : ExprL)
S_term          = (sign : Sign, term : Simple_term)
Sign            = {-, +}
Const_val       = Id v INT v (sign : Sign, id : Id)
Stmt            = Assign v Op_designator
Assign          = (as_var : Id, expr : Expr)
If              = (if_part : If_part, th_part : Th_part,
                  el_part : El_part)
If_part         = (let_schemeL : Let_schemeL, if : Expr)
Th_part         = (let_schemeL : Let_schemeL, then : A_term)
El_part         = (else : If v {error})
```

The domains Id, Pre_id, INT are not refined here. They represent alphanumeric identifiers, predefined operation identifiers of ModPascal, and the integer values.

Note that the domain Rf_body that describes the structure of the rep_function definition is based upon domains for ASPIK and ModPascal. This indicates the central, connecting role representation objects play; their character as bridge-structure is supported by allowing subsets of the participating languages to occur. Although this introduces more complexity it is inevitable: there is no way to formalize a connection between objects and items without mentioning them!

An example for rep-objects in a concrete syntax may be found in appendix A.

4.3.3. Context-sensitive Conditions

We now state for the central domains context-sensitive conditions that define the notion of static correctness for that domains. The presentation of denotational semantics in the next section will assume correct objects.

We will express the conditions as precise as possible. To do this we use auxiliary operations that are given first.

Shift moves an identifier of an identifier list to its beginning.

4.3.3.-1 Def. [Shift]
The operator shift: IdL x Id ⟶ IdL is defined by:

$$\begin{aligned}&\text{shift}((id_1, \ldots, id_n), id) := \underline{\text{if}}\ n = 1\ \underline{\text{then}}\ (id_1)\ \underline{\text{else}}\\ &\quad \underline{\text{let}}\ j := \iota\ k \in (n)\ .\ \text{for all}\ i \in (k)\ .\\ &\qquad id_i \neq id\ \underline{\text{and}}\ id_k = id\ \underline{\text{in}}\\ &\quad \underline{\text{if}}\ j = \perp\ \underline{\text{then}}\ (id_1, \ldots, id_n)\ \underline{\text{else}}\\ &\qquad (id_k, id_1, \ldots, id_{k-1}, id_{k+1}, \ldots, id_n)\end{aligned}$$

■

Objmap extracts the contents of the connect component of a rep-object hierarchy.

4.3.3.-2 Def. [objmap]
The operator objmap: Repobj ⟶ $\mathcal{P}$(Id x Id) is defined by

$$\begin{aligned}&\text{objmap}(r) := \underline{\text{let}}\ s := (\text{source (connect } r))\\ &\qquad\qquad t := (\text{target (connect } r))\ \ \underline{\text{in}}\\ &\qquad \underline{\text{if}}\ (\text{useL } r) = \perp\ \underline{\text{then}}\ \{(s, t)\}\ \underline{\text{else}}\\ &\qquad\quad \underline{\text{let}}\ (r_1, \ldots, r_n) := (\text{useL } r)\ \ \underline{\text{in}}\\ &\qquad\qquad \{(s, t)\} \cup \text{objmap}(r_1) \cup \ldots \cup \text{objmap}(r_n)\end{aligned}$$

■

A resulting mapping om can be applied to an identifier list by apply:

4.3.3.-3 Def. [apply]
The operator apply: (Id x Id) x IdL ⟶ IdL is defined by:

$$\text{apply}(om, (id_1, \ldots, id_n)) := (om(id_1), \ldots, om(id_n))$$

■

In condition RO14213 below a predicate term?: StmtL ⟶ D_BOOL is used. Term? holds if its argument - a ModPascal statement list - is of a special structure that allows to transform it uniquely into a term. In that process occurrences of ModPascal variables are substituted from right to left by operation calls, where no distinction is made between procedures, functions or initials.
The stmtL has to have this property, otherwise the surrounding let-scheme would be ill-defined (if (semantically) no term is assigned to the let-variable). The possibility of having ASPIK terms as well as ModPascal statement lists as let-scheme bodies gives the programmer of the repobjects the expressibility of both languages in the let-schemes of rep-objects.

4.3.3.-4 Def. [term?]
Let $(stmt_1, ..., stmt_n) \in StmtL$, $stmt_n$ a procedure call.
Then the predicate term?: StmtL ⟶ Bool is defined as

```
term?((stmt_1, ..., stmt_n)) :=
  if conv_lst((stmt_1, ..., stmt_n-1), conv_call(stmt_n)) ≠ ⊥
    then true else false
where
(a) conv_lst: StmtL x A_term ⟶ A_term
    conv_lst((stmt_1, ..., stmt_n) term) :=
      if n = 0 then term else
        case stmt_n ∈ Assign :
          if term contains no occurrence of (ass_var stmt_n)
            then ⊥ else
              let term_1 := term<(ass_var stmt_n) ↤
                                        (expr stmt_n)> in
                    conv_lst((stmt_1, ..., stmt_n-1), term_1)
        case stmt_n ∈ Op_designator :
          if term contains no occurrence of (var_id stmt_n)
            then ⊥ else
              let term_1 := term<(var_id stmt_n) ↤
                                    conv_call(stmt_n) in
                    conv_lst((stmt_1, ..., stmt_n-1), term_1)
(b) conv_call: Proc_stmt ⟶ A_term
    conv_call(p_stmt) :=
      case p_stmt ∈ Op_designator :
        let term_0 := (var_id p_stmt),
            (op_1, ..., op_a) := (op_idL p_stmt),
            (pl_1, ..., pl_a) := (paramLL p_stmt)  in
          let term_i for i ∈ (a) be defined as:
            case op_i ∈ Id and pl_i = ⊥ :
              term_i := (op_i term_i-1)
            case op_i ∈ Id and pl_i = (expr_1, ..., expr_n) :
              term_i := (op_i term_i-1 expr_1 ... expr_n) in
                term_a
      case p_stmt ∈ Simple_term : p_stmt
                                                          ¤
```

Remark: The definitions of conv_lst and conv_call are based on the Op_designator feature of ModPascal which allows to juxtapose several procedure and function calls in a single construct and which assigns a meaning by left-to-right evaluation of the call sequence; see [Olt 84a] and [Olt 84b] for details ('extended dot notation').
The definition of conv_lst and conv_call are derived from the equally named operators of [BR 85].

Let r ∈ RepObj. Then its context-sensitive correctness is defined as follows:

R01: r correct :⟺ (rob_id r) is unique in the environment of r and (connect r) is correct and (useL r) is correct and (operationL r) is correct and (rep_fct r) is correct.

R011: cn := (connect r) correct :⟺ (source cn) is a correct and visible module or enrichment object and (target cn) is a correct, visible, specterm-free and algorithmic spec object and
case (source cn) is module : (target cn) is one-sorted
case (source cn) is enrichment : (target cn) is zero-sorted

R012: uL := (useL r) correct :⟺ Every used rep-object is visible and correct and no cyclic usage of rep-objects occurs and for all rob ∈ 𝒫(uL) . (source (connect rob)) ∈ 𝒫(useL (source cn)) ∧ (target (connect rob)) ∈ 𝒫(useL (sp_head (target cn))) and no object is used by (source cn) or (target cn) that is not involved in some rob ∈ 𝒫(uL) and the signature morphisms of elements of uL are pairwise compatible

R013: oL := (operationsL r) correct :⟺ let
$(aop_1, \ldots, aop_n)$:= (opsL(sp_head(target cn)))
$(cop_1, \ldots, cop_n)$:= (publicL(source cn))
Op_sel := {proc_id, func_id, init_id} in
(a) for all op ∈ oL . let
aop := (from op), cop := (to op) in
∃̂ i ∈ (n) . aop = (op_id aop_i) and ∃̂ i ∈ (m) ∧
∃ oid ∈ Op_sel . cop = (oid cop_i)
(b) let aop_i, cop_j satisfy (a) for op ∈ oL in
let aopar := shift(makelist((arity aop_i)), target(cn)) in
let resob := if (result cop_j) ≠ ⊥ then (result cop_j) else (source cn),
copar := apply(objmap(r), concat((paramL cop_j), resob)) in
for all i ∈ ℕ . (first($rest^{i-1}$ aopar)) = (first($rest^{i-1}$ copar))
(c) for all j ∈ (n) . ∃̂ i ∈ (length oL) . (op_id aop_j) = (from(first($rest^{i-1}$ oL)))
(d) let oL = $(op_1, \ldots, op_a)$ in
for all i, j ∈ (a), i ≠ j .
if (from op_i) = (from op_j) then (to op_i) = (to op_j)

R014: rfct := (rep_fct r) correct :⟺
case (source cn) is enrichment : rfct = ⊥
case (source cn) is module :
(rf_id rfct) = (rob_id r) and (paramL rfct) is correct and (rf_body rfct) is correct

R0141: pL := (paramL rfct) correct :⟺
let $(vdcl_1, \ldots, vdcl_n)$:= (local_varL(local(source cn))),
$(v_1, \ldots, v_m)$:= concatL((idL $vdcl_1$), concatL((idL $vdcl_2$), ..., concatL((idL $vdcl_{n-1}$), (idL $vdcl_n$)) ...) in

let pL = (p_1, ..., p_a) in a = m and v_i = p_i for i ∈ (m)
RO142: rfb := (rf_body rfct) correct :⟺ case rfb ∈ If : (if_part rfb) is correct and (th_part rfb) is correct and (el_part rfb) is correct otherwise (let_schemeL rfb) is correct and (a_term rfb) is correct
RO1421: ip := (if_part rfb) correct :⟺ (let_schemeL ip) is correct and (if ip) is correct
RO14211: (lts_1, ..., lts_n) := (let_schemeL ip) correct :⟺ for all i ∈ (n) . let lid_i := (let_var lts_i), $lbdy_i$:= (let_body lts_i) in lid_a ≠ lid_b for a, b ∈ (n), a ≠ b and lid_i ∉ $\mathcal{P}$((v_1, ..., v_n)) [v_j as in RO141] and (case $lbdy_i$ ∈ A_term: RO14212 holds case $lbdy_i$ ∈ StmtL: RO14213 holds)
RO14212: at ∈ A_term correct :⟺ case at ∈ (at_id: Id, exprL: ExprL) : (at_id at) is visible operation identifier in the hierarchy spanned by (target cn) and occurring variables are visible let-variables, or contained in (v_1, ..., v_n) of RO141 and every sort of the arity of at is visible (if at operation) and every expression of (exprL at) is correct case at ∈ (a_id: Id, a_termL: A_termL) : [first two conditions of the first case] and every element of (a_termL at) is correct
RO14213: stmtL correct :⟺ let stmtL = ($stmt_1$, ..., $stmt_n$) in $stmt_i$, i ∈ (n) is correct and $stmt_n$ is procedure call and term?(stmtL) holds
RO14214: stmt ∈ Stmt correct :⟺ case stmt ∈ Assign : (as_var stmt) is neither visible let_variable nor contained in (v_1, ..., v_n) of RO141 and (expr stmt) is correct case stmt ∈ Op_designator : let id := (var_id stmt), (op_1, ..., op_n) := (op_idL stmt), (pl_1, ..., pl_a) := (paramLL stmt), lvL := (local_varL(local(source cn))), m := length(lvL) in let t_i = (type(first($rest^{i-1}$ lvL))), i ∈ (m) in id is visible module identifier and op_i is contained in the exported interface of some t_j, i ∈ (a), j ∈ (n) and op_a is module procedure and let pl_i = ($expr_1$, ..., $expr_{n(i)}$) in $expr_i$ is correct, i ∈ (n_i), n_i ∈ ℕ
RO14215: expr ∈ Expr correct :⟺ let t_i be defined as in RO14214 in all occurring operations are

contained in the exported interface of some t_i and all occurring variables are either let-variables or contained in (v_1, ..., v_n) of RO141
RO1422: (if ip) correct :⟺ RO14215 holds and (if ip) is boolean expression and if op ∈ Op_designator with (op_1, ..., op_n) := (op_idL op) occurs in (if ip) then op is function call
RO1423: tp := (th_part rfb) correct :⟺ (let_schemeL tp) is correct and (then tp) satisfies RO14212 and let s denote the target of (at_id tp) in S is introduced in (target cn)
RO1424: ep := (el_part rfb) correct :⟺ case op ∈ If : (if_part ep) is correct and (th_part ep) is correct and (el_part ep) is correct case ep = ⟨ERROR⟩ : true

Remarks: a) We ambiguously denote the object and the introduced data by the same identifiers ((source cn) and (target cn)).

b) The notions of specterm-free, algorithmic, one- and zero-sorted specs (RO11) are introduced in sec. 4.3.1.

c) The correctness of uL implies a hierarchical, bottom-up correctness. Compatibility of signature morphisms means that appropriate prefixing is done if operation identifiers in different specs are named identical.

d) RO13 is equivalent to: "(operationsL r) is a signature morphism". The modifications of arities in RO13 (b) are performed for the following reason: an abstract (as well as a concrete) operation may have two or more arguments of sort (target cn) ((source cn)). None of them is specially emphasized. From the view of procedural PLs, the concrete operation has to be invoked on a specific incarnation, and only this structure will be affected by possible modifications whereas other parameters are called by value. ModPascal has introduced a standard: the left-most parameter type of every module operation is of type (source cn).
To compare ASPIK operation arities and ModPascal operation arities by signature morphisms, the former are modified: the left-most occurrence of (target cn) in an operation arity is shifted to the very left position of this arity. By this, the syntactical criteria of the signature morphism condition will not fail because of ModPascal standards.
Note that the arrangement of parameter types in the arity of operations is 'syntactic sugar' and does not influence the mathematical function

behind or computational properties.

e) In the case of enrichments, (rep_fct r) is not defined (R014). This makes sense, since by definition enrichments do not introduce any data but enlarge a set of operations on already existing data. In this case the object (target cn) has to be zero-sorted (see R011).

f) Parameters of the rep-function are the local variables of the module. Note that this is coincident with the fact that carriers of module definitions are generated as cartesian products of the types of the local variables.

g) Rep-functions are either defined by A_terms or by nested elements of If.
The A_term elements directly represent values of the abstract carrier of (target cn). The if-schemes branch for different values of the local variables, and then yield in rep-function recursions or A_terms. Note that A_term is a domain that contains elements built from
   - ASPIK operation and variable symbols
   - ModPascal operation and variable symbols
   - Recursive calls of the rep-function and calls of already defined rep-functions.

   A_term elements are the most low-level syntactic items in which the connection of the different language levels can be specified and made visible.

h) Let-schemes may introduce variables with binding to ASPIK terms as well as to ModPascal statement lists. The second alternative was introduced to deal with the following situation:
   In a recursive rep-function call some parameter (a local variable) of a module type has to be modified. This modification is performed by a module procedure call on this variable.
   Now, syntactically it is impossible to write a statement on a parameter position (where expressions are expected).
   Therefore, this let-mechanism together with the syntactical checks of R014211,-13,-14, and definition 4.3.3.-4 were taken to solve the problem.

i) The domain Expr of expressions denotes pure ModPascal expressions.

j) The standard exit of the if-scheme sequence representing a rep-function definition is the ERROR-elsepart. It indicates that all concrete data elements up to now not considered are mapped to the bottom element of (target cn), i.e. that no abstract representation should exist for these concrete representations.

4.3.4. Semantics of Rep-Objects

The dynamic semantics of rep-objects is given on the syntactic domains of sec. 4.3.1., whereas the semantic domains are those already introduced in sec. 3.3.1. for ASPIK and ModPascal. In addition to the also assumed semantical functions of sec. 3.3.2. we here use a special operator for rep-objects. Finally, we assume the semantic clauses of sec. 3.4. valid.

We extend the domain Constr of all domains:

```
Constr = Spec + Sp_head + Op + ... +
         Module_type + Public + ... +
         Repobj + Connect + Operation + ...
```

Beside the general semantics function M : Constr ⟶ State ⟶ State and its derivatives E, Mt, Mm, Me, Mi we now introduce for

c ∈ Repobj

Mr : Repobj ⟶ State ⟶ State

and M⟦c⟧ϱσ ⇒ Mr⟦c⟧ϱσ.

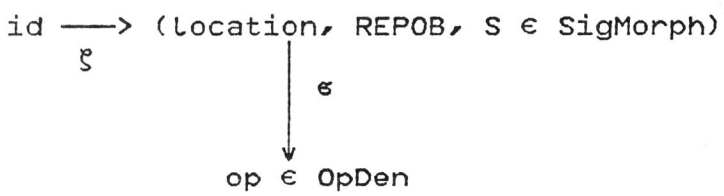
That means that we assume an environment in which ASPIK specs, ModPascal modules and rep-objects are admissable, equally entitled objects - a non-standard data base system, as for example realized in the ISDV-System (RL-DMS; see [RL 85]).
For rep-function bodies we apply E since they are purely functional:

c ∈ Rf_body

M⟦c⟧ϱσ ⇒ E⟦c⟧ϱσ

The memory model given in sec. 3.3.3. is now extended to rep-objects. Their main semantical components - the signature morphism and the rep-function - are administrated in different slots:

ϱ(id)↓2 = REPOB



The domain SigMorph is given in definition 4.3.4.- below.

Analogously to the notions of cta-environment, mod-environment and enr-environment we now introduce special environments according to rep-objects. The main requirement is that predefined syntactical operators are accessable that provide information about the syntactical object that generated a given meaning. This is necessary because our semantical domains offer no provision for this information; once a rep-object meaning is computed, information of origin of identi-

fiers or of hierarchical relations is obliterated despite the fact that it is important lateron (mainly because set union has no 'memory').

4.3.4.-1 Def. [rep-environment]

Let RClauseOps ⊆ Id with
RClauseOps := { rob?, ruse?, connect?, operations?, rf_ar?, rf_def?}

Then ϱ ∈ Env is called rep-environment if for all x ∈ RClauseOps
a) ϱ(x)≠⊥
b) ϱ(x)↓2 = REPSEL

¤

Remark: Associated to every element el of RClauseOps there is an (ambiguously denoted) special function el that evaluates to syntactical information if applied to rep-object identifiers:

el = rob?
Associated operation: rob? : Id ⟶ State ⟶ D_BOOL
ϱ(rob?) = (loc, ROBSEL, ⊥)
σ(loc) = {(id,tr)| id ∈ Id, tr ∈ D_BOOL}
rob?(id)ϱσ := σ(ϱ(rob?)↓1) (id)

el = ruse?
Associated operation: ruse? : Id ⟶ State ⟶ IdL
ϱ(ruse?) = (loc, ROBSEL, ⊥)
σ(loc) = {(id, $(id_1, \ldots, id_n)$)| id, id_i ∈ Id, i ∈ (n), n ∈ ℕ}
ruse?(id)ϱσ := σ(ϱ(ruse?)↓1) (id)

el ∈ {connect?, operations?}
Associated operation : el : Id ⟶ State ⟶ ℙ(Id x Id)
ϱ(el) = (loc, REPSEL, ⊥)
σ(loc) = {(id, $\{(id_1, id_1'), \ldots, (id_n, id_n')\}$| id, id_i, id_i' ∈ Id, i ∈ (n), n ∈ ℕ}
el(id)ϱσ = σ(ϱ(el)↓1) (id)

el = rf_ar?
Associated operation : rf_ar? : Id ⟶ Id ⟶ State ⟶ Arity
ϱ(rf_ar?) = (loc, ROBSEL, ⊥)
σ(loc) = {(id, ad)| id ∈ Id, ad ∈ ArDes}
$rf\_ar?(id_1,id_2)ϱσ := σ(ϱ(rf\_ar?)↓1)\ (id_1)\ (id_2)$

el = rf_def?
Associated operation : rf_def? : Id ⟶ State ⟶ OpDen
ϱ(rf_def?) = (loc, ROBSEL, ⊥)
σ(loc) = {(id, opden)| id ∈ Id, opden ∈ Opden}
rf_def(id)ϱσ := σ(ϱ(rf_def?)↓1) (id)

¤

Based on rep-environments, the operators EXTEND and EXT of sec. 3.4.1. are defined analogously.
Our classification of environments is used to define those environments in which we want to consider semantical defini-

tions and correctness issues: verification (v-) environments:

4.3.4.-2. Def. [v-environment]
Set ξ ∈ Env.
ξ is called verification (v-) environment, if ξ is cta-environment, mod-environment, enr-environment and rep-environment.

¤

The domain SigMorph - informally introduced in sec. 3.3.1.- serves as semantics for signature morphisms defined by rep-objects (i.e. morphisms between specifications and modules).

4.3.4.-3 Def. [SigMorph]
Let (ξ,σ) ∈ State with ξ v-environment.
The domain SigMorph of signature morphisms is defined by:
(a) SigMorph ⊆ (Map x Map x ArDes) with
for all sm ∈ SigMorph .
let f:=sm↓1, g:=sm↓2, h:=sm↓3 in
(a.1) ∀id ∈ source(f) . cta?(id)ξσ = true and
∀id ∈ target(f) . malg?(id)ξσ = true
(a.2) $\forall id_1 \in source(g)$. $\exists\ id_2 \in source(f)$.
$id_1 \in p\_op\_id?(id_2)\xi\sigma$ and
$\forall id_1 \in target(g)$. $\exists id_2 \in target(f)$.
let S := p_proc_id?(id_2) ∪ p_func_id?(id_2)
∪ p_init_id?(id_2) in
$id_1 \in S$
(a.3) source(h) = source(g) ∪ target(g)
(a.4) ∀id ∈ source(g) .
let (id_1 ... id_n, id_{n+1}) := h(id),
(id_1' ... id_m', id_{m+1}') := h(g(id)),
n,m ∈ ℕ in
(a.4.1) n=m
(a.4.2) $id_i \in source(f)$, $id_i' \in target(f)$, $i \in (n+1)$
(a.4.3) $id_i' = f(id_i)$, $i \in (n+1)$
(b) SigMorph is maximal with (a)

¤

Remark This characterization of SigMorph coincides with the signature morphism definition in 4.2.-2.

4.3.4.-4 Def. [is-sigmorph]
The predicate
is-sigmorph : (Map x Map x ArDes) ⟶ D_BOOL
is defined by

$$\text{is-sigmorph}((f, g, h)) := \begin{cases} \text{true if } (f, g, h) \in \text{SigMorph} \\ \text{false otherwise} \end{cases}$$

¤

It will be important to unite signature morphisms. To define the union we unite mappings.

4.3.4.-5 Def. [union]
Let M := (Map + ArDes), m_1, m_2 ∈ M.

Let + : M x M ⟶ (Id x (Id x Arity)) be defined as
$+(m_1, m_2) := \{(x, m_i(x)) \mid x \in source(m_i), i \in \{1, 2\}\}$.

Let $sm_i \in SigMorph$, $sm_i = (f_i, g_i, h_i)$, $i \in \{1, 2\}$.
Let $sm := (+(f_1, f_2), +(g_1, g_2), +(h_1, h_2))$.

Then sm is called the union of sm_1 and sm_2 if
(a) $sm\downarrow i$ denotes a function, $i \in (3)$
(b) is-sigmorph(sm) = true

¤

Notation: If the union of sm_1 and sm_2 is defined we ambiguously denote it by $sm_1 + sm_2$.
$sm_1 + \ldots + sm_n := (\ldots (sm_1 + sm_2) + \ldots) + sm_n)$

4.3.4.-6. Def. [signature classifications]
Let M denote a module/enrichment object.
Let $(\varsigma, \sigma) \in State$ with ς v-environment such that M is elaborated in (ς, σ). Let $\{u_1, \ldots, u_n\} := muse?((mod\_id\ M))\varsigma\sigma$
The signature
$i\Sigma(Malg(M)) := (\ldots(\Sigma(\sigma(\varsigma(u_1)\downarrow 1)) + \Sigma(\sigma(\varsigma(u_2)\downarrow 1))) + (\ldots$
$\ldots) + \Sigma(\sigma(\varsigma(u_n)\downarrow 1)))$
is called imported signature of M.
$e\Sigma(Malg(M)) := \Sigma(Malg(M))$
is ambiguously denoted exported signature of M
The tuple
$new(M) := (e\Sigma(Malg(M))\downarrow 1 \setminus i\Sigma(Malg(M))\downarrow 1,$
$e\Sigma(Malg(M))\downarrow 2 \setminus i\Sigma(Malg(M))\downarrow 2)$
is called the set of new object and operation identifier introduced by M.
Analogous definitions hold for spec objects. ¤

Remark: The union is always defined because objects (and their hierarchies) are assumed to be correct.

Now we are ready to state the semantics of rep-objects:

Sem_7 : Rep-Objects

```
MrIr: Repobj[[ς σ :=
(1)   let rid := (rob_id r), sob := (from (connect r))
          tob := (to (connect r)), (u1, ..., un) :=
          (useL r), (p1, ..., pm) := (operationL r),
          (l1, ..., la) := (paramL rf),
          rf := (rep_fct r) in

(2)   let (ς0, σ0) := EXTEND(r)ςσ in

(3)   let smi := ς(ui)↓3, i ∈ (n) in
      let sm = sm1 + ... + smn in
      if sm = ⊥ then ⊥ else

(4)   let aopi := (from pi), copi := (to pi), i ∈ (m) in
      if {aopi | i ∈ (m)} ≠ p_ops_id?(tob )ς0σ0 then ⊥ else
      let aopari := p_op_ar?(tob, aopi)ς0σ0 in
```

```
        let copar_i :=
                case cop_i ∈ p_proc_id?(sob)ξ_0σ_0 :
                                        p_proc_ar?(sob, cop_i)
                case cop_i ∈ p_func_id?(sob)ξ_0σ_0 :
                                        p_func_ar?(sob, cop_i)
                case cop_i ∈ p_init_id?(sob)ξ_0σ_0 :
                                        p_init_ar?(sob, cop_i),
                        i ∈ (m)      in
        let f denote the mapping f : {sob} ---> {tob}
            f_1 := +(sm↓1, f)        in
        if f_1 is not a function then ⊥ else
        let aopar_i = (id_i(1), ... id_i(n(i)), id_i(n(i)+1)),
                        i ∈ (m) in
        let saopar_i := (f_1(id_i(1)) ... f_1(id_i(n(i))),
                        f_1(id_i(n(i)+1))), i ∈ (m) in
        if saopar_i ≠ copar_i, i ∈ (m) then ⊥ else
(5)     let obmap := f_1,
               opmap := +(sm↓2, {(aop_i, cop_i)| i ∈ (m)}),
               armap := +(sm↓3, {(aop_i, aopar_i)| i ∈ (m)} ∪
                                {(cop_i, copar_i)| i∈ (m)}) in

(6)     if opmap or armap denote not a function then ⊥ else
        let sigmorph := (obmap, opmap, armap)   in

(7)     let rfid := (rf_id rf), rfbody := (rf_body rf)   in
        let rfdef := fix f . λξ_1σ_1 .
                        (E[[rfbody]]ξ_1[rfid ↤ (loc_1, FUNC, ⊥)]
                                σ_1[loc ↤ f])↓1
              where (ξ_1, σ_1) is (ξ_0, σ_0) but contains l_i,
                    i ∈ (a) evaluated and loc_i :=
                    newloc(ξ_0) in
(8)     let (ξ, σ) := (ξ_0, σ_0), loc := newloc(ξ_0)  in
        let ξ' := ξ[rid ↤ (loc, REPOB, sigmorph)]
            σ' := σ[loc ↤ rfdef]  in
            (ξ', σ')
```

Remarks: a) The distinction between context-sensitive conditions and dynamic semantics is slightly softened in Sem_7: the computation of the induced signature morphism is repeated. This is necessary since signature morphisms are important substructures of representation objects, and they form the special domain SigMorph.
No other context-sensitive conditions are rechecked.

b) The semantics of rep-objects is constructed in v-environments as follows:

(1) Identifier for important components are introduced by abstract syntax selections.

(2) Characteristic predefined (operation) identifier of a rep-environment are supplied with the syntactical information. This will be used in (3), (4) and (5) where generation and union of signature morphisms are performed.

Assignment to other predefined identifiers are given in Sem_8.

(3) The union of all used signature morphisms is generated via the '+'-operator of definition 4.3.4.-5. Only if the union is defined the rep-object semantics is declared.

(4)-(6) It is checked if the operation clause together with the result of (3) constitutes a signature morphism.

(7) The rep-function semantics is computed as least fixed point of the functional derived from the rep-function body.

(8) The new object is installed in the environment.

c) Note that the rep-object semantics does not denote an algebra.

Rep-function body semantics is given in Sem_8.
We assume:
Let r ∈ Reprobj.
Let rf := (rep_fct r), rid := (rf_id rf),
$(l_1, \ldots, l_a)$:= (paramL rf), rfbdy := (rf_body rf)
Let (ξ, σ) ∈ State such that necessary syntactical information about r is available (i.e. position (7) in Sem_7).

Sem_8: Rep-function bodies

```
(1) E⟦rfbdy: Rf_body⟧ξσ :=
      case rfbdy ∈ If : E⟦rfbdy: If⟧ξσ
      case rfbdy ∈ A_term : E⟦rfbdy: A_term⟧ξσ

(2) E⟦ifs: If⟧ξσ :=
      let (ξ1, σ1) := M⟦(let_schemeL(if_part ifs))⟧ξσ in
       if (E⟦(if(if_part ifs))⟧ξ1σ1)↓2 = true then
         let (ξ2, σ2) := M⟦(let_schemeL(th_part ifs))⟧ξσ in
              (E⟦(then(th_part ifs))⟧ξ2σ2)↓2
       else if (el_part ifs) = {error}
            then ⊥s else E⟦(el_part ifs)⟧ξσ

(3) M⟦lts: Let_schemeL⟧ξσ :=
      if (first lts) = ⊥ then ⊥ else
      if (rest lts) = ⊥ then M⟦(first lts)⟧ξσ else
         M⟦(rest lts)⟧(M⟦(first lts)⟧ξσ)

(4) M⟦lt: Let_scheme⟧ξσ :=
      let lid := (let_var lt), lbdy := (let_body lt) in
      case lbdy ∈ A_term :
        let loc := newloc(ξ) in
        let ξ1 := ξ[lid ↦ (loc, VAR, ⊥)]
            σ1 := σ[loc ↦ E⟦lbdy: A_term⟧ξσ] in
        (ξ1, σ1)
      case lbdy ∈ StmtL :
        if term?(lbdy) ≠ true then ⊥ else
          let lbdy = stmt1; ...; stmtn in
```

```
        let t := conv_lst((stmt_1, ..., stmt_n-1),
                          conv_call(stmt_n))  in
        let loc := newloc(ς)  in
        let ς_1 := ς[lid ↤ (loc VAR, ⊥)]
            σ_1 := σ[loc ↤ E⟦t: A_term⟧ςσ]   in
          (ς_1, σ_1)

(5) E⟦at: A_term⟧ςσ :=
      case at ∈ (at_id: Id, exprL: ExprL) :
        let aid := (at_id at),
            (e_1, ..., e_n) := (exprL at)  in
        let ev_i := (E⟦e_i⟧ςσ)↓2, i ∈ (n)  in
       ((ς, σ), σ(ς(aid)↓1)(ev_1, ..., ev_n))
      case at ∈ (at_id: Id, a_termL: A_termL) :
        let aid := (at_id at)  in
          if (a_termL at) = ⊥ then σ(ς(aid)↓1) else
            let (a_1, ..., a_n) := (a_termL at)  in
            let av_i := (E⟦a_i⟧ςσ)↓2  in
       ((ς, σ), σ(ς(aid)↓1)(av_1, ..., av_n))

(6) E⟦ex: Expr⟧ : (see Sem_4 of [Olt 84b])
```

Remarks: a) Sem_8 lists all important clauses of the rep-function body semantics computed in a v-environment; syntactic domains which do not occur in sec. 4.3.2. can be found in [Olt 84b].

b) Specific remarks:

(1) Switches only.

(2) The sort s of $\perp_s$ corresponds to the sort introduced in the source of the corresponding rep-object r (i.e. in (from(connect r))).

(3) Letscheme-lists are elaborated iteratively.

(4) Letschemes install variable-value bindings in environments. In the case of statementlists (c.f. remark h of sec. 4.3.3.) the operators term?, conv_list and conv_call of definition 4.3.3.-4 are employed. This allows to compute the semantics of the statement list on an equivalent expression.

(5) A_terms are evaluated by application of the associated function. Note that $E⟦e_i⟧$ and $E⟦a_i⟧$ describe applications of E to ModPascal (e_i) and ASPIK/rep-function (a_i) constructs; because of our choice of semantical domains and semantical functions these formulae are defined and sensefull.

(6) The domain Expr is the associated ModPascal domain.

4.3.5. Connection to Algebra Homomorphisms

Rep-objects include two essential informations: the signature morphism and the rep-function. Both are installed in v-environments as result of M. How do the so-described objects relate to algebra homomorphisms of sec. 4.2.?

For a direct application of definition 4.2.-16 we must introduce additional requirements for rep-objects to satisfy conditions (2) and (3) there. They mainly ensure that for every object of the source and target hierarchy there is an appropriate rep-object visible.

4.3.5.-1 Def. [structure respecting]
Let (ς, σ) denote a v-environment.
Let $r \in$ Repobj, and $(\varsigma_1, \sigma_1) := M[\![r]\!]\varsigma\sigma$ such that
$\varsigma((\text{rob\_id } r))\downarrow 3 \neq \perp$
Then r is called structure respecting if it holds:
Let $A_1 := M[\![(\text{from}(\text{connect } r))]\!]\varsigma\sigma$,
$A_2 := M[\![(\text{to}(\text{connect } r))]\!]\varsigma\sigma$ in
(1) $\text{Sub}(\Sigma(A_1), A_2) \neq \emptyset$
(2) $\exists\ sA_2 \in \text{Sub}(\Sigma(A_1), A_2)$.
Let (f, g) denote the signature morphism connected to sA_2 in
$(f, g) \equiv \varsigma_1((\text{rob\_id } r))\downarrow 3$
(3) $\forall\ s \in \text{use?}((\text{spec\_id}(\text{from}(\text{connect } r))))\varsigma_1\sigma_1$.
$\hat{\exists}\ m \in \text{muse?}((\text{mod\_id}(\text{to}(\text{connect } r))))\varsigma_1\sigma_1$.
$\hat{\exists}\ r' \in \text{ruse?}((\text{rob\_id } r))\varsigma_1\sigma_1$.
$(\text{from}(\text{connect } r')) = s$ and $(\text{to}(\text{connect } r')) = m$
(4) $\forall\ m \in \text{muse?}((\text{mod\_id}(\text{to}(\text{connect } r))))\varsigma_1\sigma_1$.
$\hat{\exists}\ s \in \text{use?}((\text{spec\_id}(\text{from}(\text{connect } r))))\varsigma_1\sigma_1$.
$\hat{\exists}\ r' \in \text{ruse?}((\text{rob\_id } r))\varsigma_1\sigma_1$.
$(\text{from}(\text{connect } r')) = s$ and $(\text{to}(\text{connect } r')) = m$

¤

Remark: Structure respecting is a property of rep-objects that implies isomorphic hierarchies of specs and modules/enrichments.

From this we have the following proposition.

4.5.3.-2 Proposition
A structure respecting rep-object r is a partial algebra homomorphism if its rep-function satisfies condition (4) of 4.2.-16.

Proof: We show how condition (1) to (3) of 4.2.-16 are implied.
Let A_1, A_2, sA_2, (ς, σ), (ς_1, σ_1) be as in definition 4.5.3.-1.
Let $(r_1, \ldots, r_n) :=$ closure(ruse?((rob_id r))$\varsigma_1\sigma_1$ (the closure operation generates a list of names of all directly or indirectly used objects; see [Olt 84b]).
Let $\langle rf_i : C_i \longrightarrow sC_i \mid i \in (n)\rangle$ denote a family of functions with $rf_i := \sigma_1(\varsigma_1(r_i)\downarrow 1)$, $i \in (n)$. Note that $rf_i = \perp_{OpDen}$ if for $id \in$ connect?(r)$\varsigma_1\sigma_1$ enr?(id)$\varsigma_1\sigma_1$ holds (equivalent to: sorts?(id)$\varsigma_1\sigma_1 = \emptyset$).
Then it follows:
(1) : Since r is structure respecting there exists a subalgebra sA_2 of A_2 with the required signature morphism (f, g). The direction of f (source: spec, target: module/enrichment) is invertible because of its bijectivity. Therefore in the sequel we assume f mapping

analogous to the rf_i.

(2.1) : It holds: $f(\text{obname-}A_1(C)) = \text{obname-}sA_2(sC)$ for $C \in \{C_1, \ldots, C_n\}$, $sC \in \{sC_1, \ldots, sC_n\}$.
$sC_i \notin (sA_2)\downarrow 1 \Rightarrow \text{obname-}sA_2(sC_i) \notin \text{target}(f)$
$\Rightarrow$ f is not bijective, in contradiction to the assumption.

(2.2) : Again by the bijectivity of f

(3) : If $c_a \in (A_1)\downarrow 1$, but no i $\in$ (n) exists with $c_a = c_i$
$\Rightarrow$ there is a spec S not involved in some rep-object r_i
$\Rightarrow$ the object mapping f is not total on the set of visible objects, contrary to the assumption.

If additionally (4) is satisfied by $\langle rf_i: C_i \longrightarrow sC_i \mid i \in (n)\rangle$ the proposition follows directly. ¤

This relation between structure respecting rep-objects and partial algebra homomorphisms will be used for the formulation of our correctness criteria in the next section.

4.4. Realization Conditions

We now return to our treatment of realizations and realization contexts. Both notions differ only in the additional requirements demanded for realizations.

The situation is the following: there are a user-defined spec object, a module/enrichment object and a rep-object. The first two are semantically described by algebras; the rep-object establishes a signature morphism between them, and under certain conditions an algebra homomorphism. These additional conditions will be embodied in the realization re-definition of 4.1.-2.

4.4.-1 Def. [realization, realization-context]
Let S denote an ASPIK spec, M a ModPascal module/enrichment and R a rep-object.
Let C_1 denote the conditions
- S is algorithmic, single-sorted, specterm-free
- M is instantiation type definition free
- R is defined on S and M
- R is structure respecting

Let C_2 denote the condition
- R satisfies (4) of 4.2.-16

Then the triple (S, M, R) is called realization, if its components satisfy C_1 and C_2. It is called realization-context, if only C_1 is satisfied.

¤

By this definition, a user of a software development system has to proceed in three steps to verify his sequential implementation (=realization):

1) Specify the task in constructive, at most one-sorted specterm-free spec objects. Re-program it in modules and enrichments while utilizing efficiency increasing features.

2) Write a rep-object in which you try to express the intended relations syntactically (signature morphism) and semantically (carrier mapping).
   Compute the mapping between the canonical term algebra and the module algebra induced by the rep-object.
3) According to definition 4.2.-16, show for each operation of the canonical term algebra, that the homomorphism equations holds.

In section 5 we will develop a proof method suitable to this steps.

4.4.-2. Def. [correctness, realization conditions]
A module (enrichment) M is said to realize a specification S correctly, if there exists a rep-object R such that (S, M, R) is a realization. The homomorphy equations derived from R (the set C_2 of 4.4.-1) are called realization conditions.

¤

Note that this definition of correctness depends not on a specific rep-object (perhaps several will do it). But it is obvious that correctness statements for fixed S and M, and for different R come up with incomparable semantical structures. Therefore realizations (S,M,R_1) and (S,M,R_2) are not exchangeable in general.
Note also, that there are no limitations in the number of realization (context)s an object may be involved. This is solely an administration problem which has to be resolved by the object management of the software development system.

One may argue that the homomorphism property is too weak to serve as correctness criteria for refinements of this kind. We do not believe this. If a designer of software can be assured that his final program behaves in just the way he specified on the abstract level, he will be satisfied and not be worrying about the possibility that it might do more than he intended. If additionally homomorphy is easier derivable in practical environments than say isomorphy one should not feel uncomfortable with this alleged weakness.

In section 4.1. the problems arising from non-surjective carrier mappings were discussed. In the special context of realizations surjectivity is delivered for-free.

4.4.-3. Corrollary
Let (S, M, R) denote a realization with homomorphisms $\langle rf_i: C_i \longrightarrow C_i' \mid i \in (n)\rangle$ for some n.
Then rf_i is surjective, $i \in (n)$.

Proof: Let (ξ, σ) denote a v-environment with semantical embedding of S, M, and R, such that the homomorphisms of the premise exist as $\sigma(\xi(r_i)\downarrow 1)$ for some rep-objects r_i, $i \in (n)$.

Let sid := sort_id?((spec_id S))$\xi\sigma$, $C_S := \sigma(\xi(sid)\downarrow 1)$,
$\{con_1, \ldots, con_a\}$:= constr?((spec_id S))$\xi\sigma$,
C_M := toi?((mod_id M))$\xi\sigma$.

Let $(rf: C_M \longrightarrow C_S) := \iota\ (rf_j: C_j \longrightarrow C_j') \in \langle rf_i: C_i \longrightarrow C_i' \mid i \in (n)\rangle\ .\ C_M = C_j$ and $C_S = C_j'$

Then it is sufficient to consider the surjectivity of rf since r is structure respecting.

Let $SIG_i = (OB_i, OP_i, arity_i)$, $i \in \{M, S\}$ denote the associated signatures to S and M.
Let $(f, g) : SIG_S \longrightarrow SIG_M$ denote the restriction of $\xi((rob\_id))\downarrow 3$ to SIG_S.

(1) Every element of C_S is finitly generatable by applications of con_i, $i \in (a)$
$\Rightarrow$ structural induction is applicable

(2) $\forall\ con \in \{con_1, \ldots, con_a\}\ .$
$arity_S(con) = (\varepsilon, sid) \Rightarrow$
$con = rf(g(con))$
since rf is homomorphism.

(3) $\forall\ con \in \{con_1, \ldots, con_a\}, (arity_S(con))\downarrow 1 \neq \varepsilon\ .$
$\forall\ sid_1 \ldots sid_b \in OB_S^*, b \in \mathbb{N}\ .$
$\forall\ t_i \in C_i' (\equiv \sigma(\xi(sid_i)\downarrow 1))\ .$
Induction hypothesis: $t_i = rf_i(g(t_i))$
(where g is applied to the constituents of t_i)
Induction step: $arity_S(con) = (sid_1 \ldots sid_b, sid) \Rightarrow$
$con(t_1, \ldots, t_b) = rf(g(con(t_1, \ldots, t_b)))$
since rf is homomorphism
Every element of C_S is target under rf; from this conjecture follows.

¤

The surjectivity of the homomorphisms induced by rep-objects (S, M, R) allows to factorize the module algebra by a congruence induced by the homomorphism. The factor algebra is isomorphic to CTA(S) and to the relevant subalgebra of M(ALG) (where relevant means: with respect to the correctness issue in SEEs). The development of this results rounds this section off.

4.4.-4. Def. $[\equiv_{rf}]$
Let (S, M, R) denote a realization with $\langle rf_i: C_i \longrightarrow C_i' \mid i \in (n)\rangle$.

Then the carrier identification $\equiv_{rfi}$ induced by rf_i is defined as:

$$\equiv_{rfi} := \{(c_1, c_2) \mid c_1, c_2 \in C_i \text{ and } rf_i(c_1) = rf_i(c_2)\}$$

¤

Notation: $\equiv_R := \{\equiv_{rfi} \mid i \in (n)\}$
$[c] := \iota \equiv_{rfi} \in \equiv_R\ .\ \exists (c_1, c_2) \in \equiv_{rfi}\ .$
$c_1 = c$ or $c_2 = c$

In the next lemma the notion of a Σ-congruence is employed where Σ denotes a signature. A Σ-congruence relation is congruence relation that holds for all operations named by ele-

ments of Σ.

4.4.-5. Lemma
Let (S, M, R) denote a realization with $\langle rf_i: C_i \longrightarrow C_i' \mid i \in (n)\rangle$ and with associated subalgebra sM of Malg(M). Let Σ = (OB, OP, arity) := Σ(sM).

Then $\equiv_R$ is a Σ-congruence.

Proof:
(1) $\equiv_R$ is equivalence relation
(1.1) $\equiv_R$ reflexive: $\forall c_i \in C_i \;.\; rf_i(c_i) = rf_i(c_i)$
$\iff c_i \equiv_{rfi} c_i$.
(1.2) $\equiv_R$ symmetric: $\forall c_1, c_2 \in C_i \;.\; c_1 \equiv_{rfi} c_2$
$\iff rf_i(c_1) = rf_i(c_2)$
$\iff rf_i(c_2) = rf_i(c_1)$
$\iff c_2 \equiv_{rfi} c_1$
(1.3) $\equiv_R$ transitive: $\forall c_1, c_2, c_3 \in C_i$.
$c_1 \equiv_{rfi} c_2$ and $c_2 \equiv_{rfi} c_2$
$\iff rf_i(c_1) = rf_i(c_2)$
and $rf_i(c_2) = rf_i(c_3)$
$\Rightarrow rf_i(c_1) = rf_i(_3)$
$\iff c_1 \equiv_{rfi} c_3$

(2) $\equiv_R$ Σ-congruence
to show: $\forall$ op $\in$ OP with arity(op) =
(obname-Malg(M)($C_{i(1)}$) ... obname-Malg(M)($C_{i(k)}$),
obname-Malg(M)(C_m)),
C_m, $C_{i(j)} \in \{C_1, \ldots, C_n\}$, i_j, m $\in \{1, \ldots, n\}$, j $\in$ (k).
$\forall$ $c_{i(j)}$, $c_{i(j)}' \in C_{i(j)}$, m, $i_j \in \{1, \ldots, n\}$, j $\in$ (k)
.
$c_{i(j)} \equiv_{rfi(j)} c_{i(j)}' \Rightarrow$
$op(c_{i(1)}, \ldots, c_{i(k)}) \equiv_{rfm} op(c_{i(1)}', \ldots, c_{i(k)}')$
Then we have
let (ξ, σ) denote a v-environment with elaborated R in
let (f, g) := ξ((rob_id r))$\downarrow$3 in
$\forall$ $c_{i(j)}$, $c_{i(j)}' \in C_{i(j)}$, m, $i_j \in \{1, \ldots, n\}$, j $\in$ (k) .
$c_{i(j)} \equiv_{rfi(j)} c_{i(j)}'$
$\Rightarrow rf_m\,(op(c_{i(1)}, \ldots, c_{i(k)}))$
$= f^{-1}(op)(rf_{i(1)}(c_{i(1)}), \ldots, rf_{i(k)}(c_{i(k)}))$
$= f^{-1}(op)(rf_{i(1)}(c_{i(1)}'), \ldots, rf_{i(k)}(c_{i(k)}'))$
$= rf_m(op(c_{i(1)}', \ldots, c_{i(k)}'))$
$\Rightarrow op(c_{i(1)}, \ldots, c_{i(k)}) \equiv_{rfm} op(c_{i(1)}', \ldots, c_{i(k)}')$
¤

In general Malg(M) contains more data and operations than are of interest (local types, local operations). In Sub(Σ(CTA(S)), Malg(M)) only those algebras occur that lack superfluous items. Since every realization is associated to a specific algebra out of this set, we take those algebras for factorization instead of Malg(M).

4.4.-6. Def. [Q(M, R)]
Let (S, M, R) denote a realization with $\langle rf_i : C_i \mathrel{—} C_i' \mid i \in (n)\rangle$ and associated subalgebra sM, (ξ, σ) $\in$ State with ξ

v-environment and elaborated R.
Let $\Sigma(sM) := (OB, OP, arity)$, $(f, g) := \xi((rob\_id\ R))\downarrow 3$
Let $[\_]$ denote the congruence classes generated by $\equiv_R$.

Then the quotient algebra $Q(M, R)$ of M by R is defined by:
$Q(M, R) := (C_q, O_q)$ where
(1) $C_q := \{C_{ob} \mid ob \in OB\}$
(2) $C_{ob} := \{[c] \mid c \in C_i$ and $obname\text{-}\Sigma(sM)(C_i) = ob$ for some $i \in (n)\}$
(3) $\forall o \in (sM)\downarrow 2$ with $arity\text{-}\Sigma(sM)(opname\text{-}\Sigma(sM)(o)) = (ob_1 \ldots ob_n, ob), ob_i, ob \in OB, i \in (n)$.
$\forall c_i \in C_i$ with $obname\text{-}\Sigma(sM)(c_i) = ob_i, i \in (n)$.
there exists an operation o_q defined by
$o_q([c_1], \ldots, [c_1]) := [o(c_1, \ldots, c_n)]$
(4) $O_q := \{o_q \mid o \in (sM)\downarrow 2\}$

¤

Remark: This defintion is independent from the choice of the $[c_i]$.

By definition 4.4.-6 there is an induced signature morphism $(\bar{f}, \bar{g}) : \Sigma(sM) \longrightarrow \Sigma(Q(R, M))$ with
(1) if $rf_i : C_i \longrightarrow C_i'$: let $ob := obname\text{-}\Sigma(sM)(C_i)$ in $\bar{f}(ob) = obname\text{-}\Sigma(Q(R, M))(C_{ob}) = ob$
(2) if $o \in (sM)\downarrow 2$: let $op := opname\text{-}\Sigma(sM)(o)$ in $\bar{g}(op) = opname\text{-}\Sigma(Q(R, M))(o_q) = op$

Since the identical morphism does cause no harm in compositions we apply (f, g) also in situations where $(\bar{f}, \bar{g}) \circ (f, g)$ is correct.

$Q(M, R)$ is the semantical object that generates the already mentioned identifications and justifies our notion of correctness:

4.4.-7 Lemma
Let (S, M, R) denote a realization with $\langle rf_i : C_i \longrightarrow C_i' \mid i \in (n)\rangle$, sM, and $Q(M, R)$.

Then
(1) $Q(M, R)$ is isomorphic to CTA(S).
(2) $Q(M,R)$ is homomorphic to sM.

Proof:
Let (f, g) denote the signature morphism associated to sM.

(1) Let $\langle is_i : C_{ob} \longrightarrow C_i' \mid i \in (n)$ and $obname\text{-}\Sigma(sM)(C_i') = ob\rangle$ be defined by
$is_i([c_i]) := rf_i(c_i)$
(1.1) is_i is injective: $\forall [c_1], [c_2] \in C_{ob}$.
$is_i([c_1]) = is_i([c_2])$
$\Longleftrightarrow rf_i(c_1) = rf_i(c_2) \Longleftrightarrow [c_1] = [c_2]$
(1.2) is_i is surjective: rf_i is surjective (c.f. 4.4.-3)
$\Rightarrow \forall c_i' \in C_i' . \exists c_i \in C_i . rf_i(c_i) = c_i'$.
Also $\forall c_i \in C_i . \exists [c] \in C_{ob} . (c_i, c_i) \in [c]$.
$\Rightarrow \forall c_i' \in C_i'$ with $c_i' = rf_i(c_i)$ for some $c_i \in C_i$.

$\exists$ [c] $\in C_{0b}$. $(c_i, c_i) \in$ [c] and $is_i([c]) = rf_i(c_i) = c_i'$.

(1.3) is_i is homomorphism: Let $(OB_1, OP_1, arity_1) := \Sigma(Q(M, R))$, $(OB_2, OP_2, arity_2) := \Sigma(CTA(S))$.
$\forall\, o_q \in (Q(M, R))\Downarrow 2$ with opname-$\Sigma(Q(M, R))(o_q) = op_1$
and $arity(op_1) = (ob_1 \ldots ob_n, ob)$
and $o \in (sM)\Downarrow 2$, opname-$\Sigma(sM)(o) = f($opname-$\Sigma(Q(M, R))(oq))$.
$\forall\, o_2 \in (CTA(S))\Downarrow 2$ with opname-$\Sigma(CTA(S))o_2) = op_2$
and $op_2 = g(op_1)$ and $arity_2(op_2) = (f(ob_1) \ldots f(ob_n), f(ob))$.
$\forall\, [c_i] \in C_{0b}, c_i \in C_i, i \in (n)$.

$is_i(o_q([c_1], \ldots, [c_n]))$
$= is_i([o(c_1, \ldots, c_n)])$ def. o_q
$= rf_i(o(c_1, \ldots, c_n)])$ def. is_i
$= o_2(rf_1(c_1), \ldots, rf_n(c_n))$ rf_i homom.
$= o_2(is_1([c_1]), \ldots, is_n([c_n]))$ def. is_i

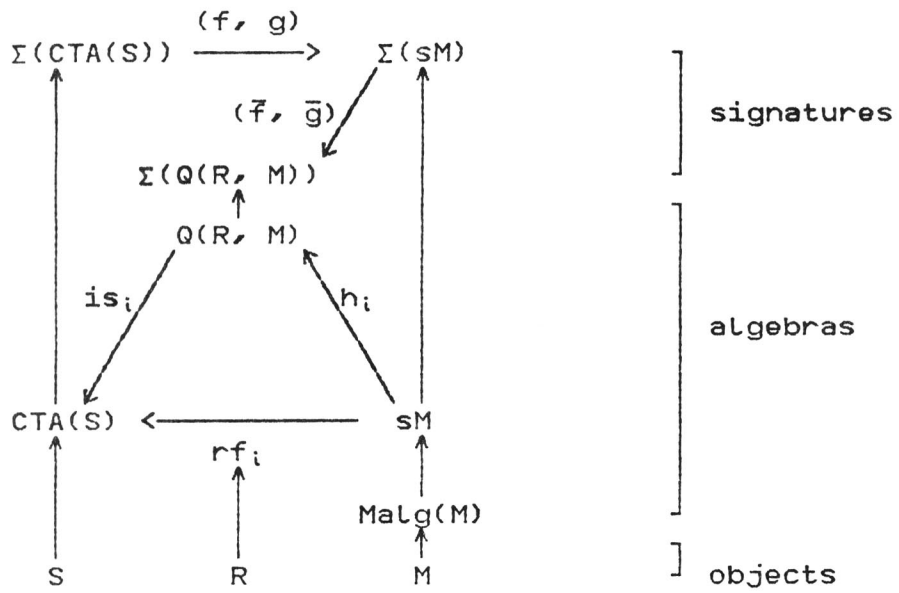
(2) Let $\langle h_i: C_i \longrightarrow C_{0b} \mid i \in (n),$ obname-$\Sigma(sM)(c_i) = ob\rangle$ be defined by:
$\forall\, c_i \in C_i$: $h_i(c_i) := [c_i]$
Let $(OB_1, OP_1, arity_1) := \Sigma(sM)$, $(OB_2, OP_2, arity_2) := \Sigma(Q(M, R))$.
$\forall\, o_1 \in (sM)\Downarrow 2$ with $arity_1($opname-$\Sigma(sM)(o_1)) = (ob_1 \ldots ob_n, ob)$ and $o_2 \in (Q(M, R))\Downarrow 2$ with $arity_2($opname-$\Sigma(Q(M, R))(o_2) = (f(ob_1) \ldots f(ob_n), f(ob))$ and opname-$\Sigma(Q(M, R))(o_2) = f($opname-$\Sigma(sM)(o_1))$.
$\forall\, c_i \in C_i$ with obname-$\Sigma(sM)(C_i) = ob_i, i \in (n)$.

$h_i(o_1(c_1, \ldots, c_n))$
$= [o_1(c_1, \ldots, c_n)]$ def h_i
$= o_2([c_1], \ldots, [c_n])$ def o_2
$= o_2(h_1(c_1), \ldots, h_n(c_n))$ def h_i ∎

Remarks: a) Note that in (1.3) (f, g) is applicable since $(\bar{f}, \bar{g}) \circ (f, g) = (f, g)$
b) Result (2) holds for every factorization by a homomorphism.

With this result we are able to provide a sufficient quality measure for our refinement scenario in SEEs. We summarize the most important structures graphically.

4.4.-8 Fig. [realization scenario]
Let (S, M, R) denote a realization.



■

The lowest level (looking at 4.4.-8 as bottom-up directed graph) represents our starting situation in SEEs: specification and module are given; a re-co is constructed by the addition of R that implies mappings rf_i and a signature morphism (f_r, g_r) (link omitted). Malg(M) is restricted to sM to consider only relevant (for the verification issue) carriers. This restriction comes with a signature morphism that has to be isomorphic to (f_r, g_r) (practically, the - with respect to R - appropriate subalgebra sM $\in$ sub(Σ(CTA(S)), Malg(M)) is chosen for the verification). If the verification conditions hold (i.e. the re-co (S, M, R) becomes a realization), the rf_i can be used to factorize sM yielding Q(R, M), an isomorphic algebra to CTA(S). Q(R, M) is semantically the 'essential chunk' of the M semantics, and of M. Its isomorphy implies intended errorfree refinement of S into M.
The signature morphisms behave in the indicated manner. The notions and results of this section are applied in a concrete method for the verification of refinements. It is given in sec. 5.

4.5. Other Approaches to Object Correctness

The problem of assigning a satisfactory semantics to the implementation issue in SEEs has been recognized in various publications. Even if not SEE application was the primary goal of the proposals almost every author claims that his solution will do it well. In the following we briefly look at different approaches having our SEE scenario as developed in sections 1 and 2 in mind.

Therefore theories for the abstract implementation are not

considered (i.e. implementation of algebraic specifications by algebraic specifications) although by far the most research activity was and is concentrated on this issue. One reason for this lies in our judgement in the elegant and powerful mathematical mechanisms that are best applicable if the considered structures are as mathematical as possible - a requirement that the harder is to meet the more concrete the structure to model is. Approaches of the ADJ group ([ADJ 78], [ADJ 79]), Ehrig et al. ([EKP 78]), Ehrich ([EL 80]), Klaeren ([Kla 82]), Poigne ([Poi 83]), Sanella/Wirsing [SW 82], or Beierle and Voss ([BV 85]), among others, fall in this category. Nevertheless these approaches have strongly influenced the efforts for a correctness notion for realizations (= concrete implementations), and all suggestions below make more or less direct access to their notions and results.

(a) Algebraic Specifications as Programming Language Semantics

Pair [Pai 80] studies the adjustment of an initial algebra semantics for a programming language and uses this concept to show compiler correctness. For this purpose he defines an abstract data Type T_L for a language L. T_L contains

sorts:
- for each primitive type p of L there is a corresponding sort p* in T_L, which is the set of values of p
- for each primitive type p of L there exists a sort p^+ which is the set of expressions of p
- a sort 'Stmt' of statements
- sorts for states, identifiers, declarations, labels etc.

operations:
- each primitive operation of L has an associated operation in T_L
- operations that simulate the behaviour of program constructs (concat : Stmt x Stmt ⟶ Stmt)
- an operation 'apply' for executing statements in a state
- an operation 'eval' that evaluates expressions in a state
- a number of auxiliary functions used to define the effect of the programming language constructs.

axioms:
- all operations are defined by more or less complex axioms of eval.

The connection between the program constructs and their abstract terms is given by a 'syntactical abstraction' function sa that is defined on the grammar and applied to the syntactic tree of a program.

In most of the sorts and operations introduced one can see a strong intention to model the language semantics in a close relationship to a denotational semantics (DS), though no fixpoint theory is actually used. But modelling stores, environments, blockstructures etc. enforces a comparison, and here the increasing complexity of the defining axioms seemed

to be not advantageous in most cases versus corresponding semantics function clauses in a DS (e.g. states are 'terms' of subsequent assignments of values to identifiers than a mapping between sets).

The result is a full description of a programming language semantics as the initial algebra semantics of a single abstract data type T_L. Thus, if a language contains a module - like construct, by the method it would correspond to some abstract sorts and operations in T_L and a programming system that is devised to check compatibility of a given abstract object ob with a module object mod defined in L could proceed as follows:

a) compute the semantics of mod in the quotient term algebra $Q_{T(L)}$.
b) compile the semantics of ob (an algebra)
c) show their isomorphy

Thus fully semantical preservation would be guaranteed.
But in general and especially in our environment this method comes with problems:

- if nothing more about the semantics of a module is said, the only choice will be a congruence class of $Q_{T(L)}$ (namely that class that contains the term associated to the module definition). Then an isomorphism between terms and algebras gives no sense (c.f. sec. 4.1.).

- If the semantics of a module is a subalgebra S of $Q_{T(L)}$ an algebra isomorphism has to be constructed by the system - a formidable task that is not realistic at the moment.

(b) Denotational Semantics for Programming Languages based on Algebra Transformations

Ganzinger [Gan 82] connects an algebraic semantics of a specification with an algebraic semantics of a module in a simple language by defining a denotational semantics for the language that uses algebra classes as domains. Type definitions (modules) are considered as in [Hoa 72] as consisting of a list of procedure or function declarations, a set of representation variables and an initialization, and a program is regarded as a list of typedefinitions.

The classical method to define states as mappings Id ⟶ Values and environments as mappings Id ⟶ Operationdenotation is substituted by giving environments as algebraic specifications u = (S, Σ, E) [s, Σ: sets of type/operation identifiers, E set of axioms of the operators] and states being algebras in the category Alg[u] generated by the environment specification. Then for example type identifiers and variable identifiers (nullary operation symbols) in u are supplied with concrete values in each state (carrierset, concrete operation) and executing a language - statement will mean a mapping between the state algebras.

To get suitable domains, the categories Alg[u] are (partially)

ordered to become a complete lattice and continuous functors between the categories are defined that correspond to the free (construction) functor F and the forgetful functor U of [ADJ 78a], here being used in the definition of enrichments and restrictions of specifications. The main step then is to define on programming language constructs C the functions

$E[\![C]\!] : U \longrightarrow U$ (U = Environments)

and

$T_u[\![C]\!] : Alg[u] \longrightarrow Alg[E[C]u]$

i.e. an environment transforming function E and a state transformation T_u for $u \in U$.

For (module) type definitions td $E[\![td]\!](u)$ is an extended algebraic specification u' in which the operation- and type names introduced in the type definition and some special operations are added to u, whereas $T_u[td](A)$ is a free algebra construction generated on (the state algebra) A and the operation symbols introduced in td. The resulting algebra gets its operations from the application of $T_u[op]$ to each operation definition of td, that itself is a model of op as an argument-result relation on the carriers associated with the parameter- and value types of the operation op.

Starting with an environment PRE of the predefined types of the language the methodology extends the initial PRE - algebra by user defined types via $T_{PRE}[typedef]$. Correctness of a program P with respect to a specification SPEC could be defined as

Let $A := I_{PRE}$ (initial PRE - algebra) in
$T_{PRE}[P](A) := I_{SPEC}$ (initial SPEC - algebra).

Being used in our system, an algorithm for checking the correctness of a realization of a specification could look like:

1) Compute the initial algebra of the predefined types of the used programming language.
2) Generate the free construction over the initial PRE - algebra and the types and operations introduced by the module definition.
3) Find an isomorphism between this free construction and the canonical term algebra (the semantical algebra of the specification).

The feasibilty of such a correctness proof seems to be highly unrealistic in a mechanical system because of sensitive tasks as initial algebra construction, free construction and isomorphism generation, which are already on theoretical level need sophisiticated mathematics that cannot be reproduced by today's proof systems. Additionally, the fact that algorithmic specifications contain more concrete information about carriers and operations than purely algebraic specifications has not been taken into account in this approach.

Nevertheless this link between algebraic specifications and

procedural programming languages has given some advice to the semantics of ModPascal.

(c) Terminal Algebra Semantics for Modules

Schulz [Sch 82] has developed a method for realizing abstract data types that is based on a terminal algebra semantics. Generating contexts (terms of sort S' with exactly one variable of sort S), applying them to all appropriate terms of the Herbrand - universe of the specification and identifing those which are undistinguishable by equational reasoning in all contexts, gives a formalization of the idea that only the outside visible behaviour of a module is important for its semantics.

This congruence generation is also applied to an algebra extracted from a module definition in a specific language that is supposed to realize the abstract specification. Correctness is then defined as the isomorphy of the associated terminal algebras and a method is given that constructs a set of "verification conditions" for each concrete operation of the module, whose fullfillment implies the isomorphy.

Taking this approach over to our system ignores two important facts:
- the semantics of an algorithmic specification is a canonical term algebra whose carriers contain elements, which by definition must not be identified with any other. Thus establishing a terminal congruence relation is contrary to the fact of being canonical.
- The operations in specifications are not defined by equations but by functional recursive schemes. Their behaviour has to be modelled in the algebra derived from the module definition. The proposed method cannot handle this.

(d) Transformation Rules as Operation Semantics

The CIP project ([Bau 81]) aimed at goals similar to our scenario: a SEE that provides for most activities semantical foundations which enable and support verification issues.

In their environment Laut [Lau 80] starts with 'computation structures' (algebras's with finitely generated carriers and a set of functions which together satisfy the axioms of the specification) of an algebraic specification and defines an associated module to the computation structure. The operations of the module are assumed to satisfy denotational transformations as

$$\frac{\text{call mod.op}(x_1, \ldots, x_n)}{x_1 := \text{op}(x_1, \ldots, x_n)} \quad \updownarrow$$

(what means that the effect of the module operation call can be modelled as an assignment of the value of an invocation of the 'abstract' function to its first argument as suggested in

[Hoa 72]). With this semantics of procedure calls he shows that the axioms of the algebraic specification are also satisfied by the module operations. Because of the use of a predicate transformer semantics and the restriction to assignments this task reduces to the comprehensable and well-known process of generating weakest preconditions for assignments.

Unfortunately, no language construct is considered whose instantiations are capable of possessing an internal state and are passed to other objects or are stored. Also the tranformation rules for the different types of arities of the abstract operations make no difference between program variables and term variables and they add assignments to a functional language. Therefore only little information can be derived from this approach w.r.t. our environment.

5. A Proof Method

A very important characteristic of correctness criteria in general is the degree of mechanization that can be achieved in order to proof their validity. If for example nearly no mechanic support is obtainable the integration of the criteria in a SEE would be senseless. On the other hand it follows from research in this area and already implemented solutions, that full mechanization is currently impossible - due to limitations of existing proof systems. As a consequence a semi-automatic procedure is a most likely candidate, and in the following we present a proof method for realizations (PMR) that involves user-dependant, method-dependant and system-dependant substeps, where the last two modes are performed automatically. We firstly introduce the substeps in sec. 5.1, point out limitations in sec. 5.2., and then assemble the substeps to a method for the proof of the realization property (PMR) applicable in SEEs (sect. 5.3.)

5.1. Basic Steps

In our scenario the check of the realization property is to mechanize as far as possible. In other words: the validity of a set of equations (homomorphy equations) in a certain theory has to be shown. Roughly there are five steps:

(1) generation of a set HEQ of homomorphy equations from a given re-co (S, M, R)
(2) involvation of hierarchy information (of S, M and R) into HEQ
(3) formulation of an induction proof task
(4) transferring proof tasks to proof systems, and
(5) administrating results in the SEE.

5.1.1. HEQ Generation

This step is primarily of syntactical nature. Given a re-co (S,M,R) the standard homomorphy equations can be generated automatically. We distinguish two cases dependant on the used formal language for HEQ:

- HEQ is multi-formal
- HEQ is single-formal

where the terms multi- and single-formal refer to situations that HEQ contains occurences of terms of more than one formal system resp. exactly one formal system. We make this distinction more precise below.

5.1.1.1. Multi-formalism

Since S, M and R are objects of the SEE data base and therefore possess correctness flags the set HEQ may be constructed automatically from the information contained in the objects (see in the appendix for an example). The general form of these equations is:

(*) $S\text{-}op(rf_1(arg_1),\ldots,rf_n(arg_n)) = rf(M\text{-}op(arg_1,\ldots,arg_n))$

where S-op, M-op denote operations of S and M that are connected by the signature morphism induced by R
rf denotes the rep-function of R
rf_i denotes rep-functions of used rep-objects of R, i $\in$ (n).
arg_i denote appropriate argument expressions of ModPascal.

The proof of HEQ bears some problems. In general, the theory in which the equations are formalized is not predicate calculus or some other standard logic, and therefore standard techniques do not apply. In general, the formal systems are:

S-op	: ASPIK operation, algorithmically defined.
M-op	: ModPascal operation, defined by an imperative program.
rf, rf_i	: carrier mappings of rep-objects, defined in a mixed ASPIK/ModPascal mode.
arg_i	: ModPascal variables and expressions, taking values out of a semantical carrier.

We call the above situation multi-formal since several formalisms are used to express HEQ.

At first glance these different items may be united by their algebraic meaning: ASPIK as well as ModPascal operations are associated to algebra operations, and the carrier mapping is easily embedded. But the semantics is defined denotational, and the meanings of operations are constructed via least fixed points of associated functionals. This does not allow reasoning in standard logic, since fixed points cannot be expressed in first order logical formulas. To be able to proof propositions under this preconditions one has to employ methods and tools capable of dealing with denotational

semantics (e.g. LCF [GMW 79]).

In that case equation (*) above would become

$$M[\![S\text{-}op(rf_1(arg_1),\ldots,rf_n(arg_n))]\!]\xi\sigma \equiv M[\![rf(M\text{-}op(arg_1,\ldots,arg_n))]\!]\xi\sigma$$

for states (ξ,σ) in which the occuring operation identifiers are defined; an unfolding of M yields to

$$\sigma(\xi(S\text{-}op)\downarrow 1)(E[\![rf_1(arg_1)]\!]\xi\sigma,\ldots,E[\![rf_n(arg_n)]\!]\xi\sigma) = \sigma(\xi(rf)\downarrow 1)(M[\![M\text{-}op(arg_1,\ldots,arg_n)]\!]\xi\sigma)$$

$$\Longleftrightarrow$$

$$(**)\quad \sigma(\xi(S\text{-}op)\downarrow 1)(\sigma(\xi(rf_1)\downarrow 1)(E[\![arg_1]\!]\xi\sigma),\ldots,\sigma(\xi(rf_n)\downarrow 1)(E[\![arg_n]\!]\xi\sigma)) = \sigma(\xi(rf)\downarrow 1)((\sigma(\xi(M\text{-}op)\downarrow 1)(E[\![arg_1]\!]\xi\sigma,\ldots,E[\![arg_n]\!]\xi\sigma)))$$

(first component selection of $E[\![arg_i]\!]\xi\sigma$ is omitted).

It is obvious that a proof of the validity of (**) for given $(\xi,\sigma) \in State$ goes beyond the scope of the currently most easily available first order theorem provers; more appropriate systems are not designed for this application and this use inside a SEE. This could call in question our approach since we appearently have to pay our employment of denotational semantics with unmechanizability of associated proof tasks.

A first answer to this objection may point at the temporal character of this situation. Since the theory behind denotational semantics is well-developped and several proof techniques are known (e.g. fixpoint induction, fixpoint computation; c.f. [Man 74]), a proof system suited to our needs could be very well implemented, with special emphasis on usability in SEEs. Here we will not further investigate this alternative.

More important is another solution that is based on the fact that under certain circumstances the set HEQ can be generated by using a single formalism. Below we make concrete this idea (sec. 5.1.1.2.).

It should be emphasized that this complications do not influence our principal conviction that denotational semantics is best-suited to describe SEE language semantics. The exactness and uniqueness of this formalism makes disambiguities impossible, and it gives every SEE user a solid framework for his software development indepently from the necessity of verification.

5.1.1.2. Single Formalism

We now present a solution to the multi-formalism problem. The set HEQ is automatically generated in the form given in the previous section, but then modified until the equations are written in a single formalism: as properties of an ASPIK

specification.

A very important fact is that this process is totally mechanical: if some pre-conditions are satisfied (essentially object-associated properties that are administrated in the data base of the SEE and therefore are easily accessable and checkable), then the modifications of HEQ take place according to a given algorithm without user interaction. The denotational semantics problems can be disregarded; it only remains to make sure that the employed descriptions for the languages as a whole coincide, i.e. that

- the equational (first order) descriptions used for ASPIK specifications and the denotational semantics for ASPIK are equivalent
- the proof theory used for ModPascal is equivalent to the denotational semantics
- the denotational semantics for rep-objects is a well-defined extension of the ASPIK and ModPascal semantics.

This is a voluminous task, but it has to be performed solitarily and independently from a given SEE scenario, e.g. by the SEE designer. It then provides an exact base for SEE languages / objects and SEE verification theories (an exemplary treatment of equivalence of various language definitions can be found in [Don 76]). For our treatment we assume that the three equivalences above are shown.

An algorithm TR for transformation of single-formalism HEQ (short: SHEQ) out of multi-formalism HEQ (MHEQ) is influenced by the fact that despite of the problems arising from the ASPIK semantics there are proof tasks of the abstract level that can be decided within standard logic. The reason is that sometimes it is not necessary to compute the algebra operation behind an algorithmic definition. Instead, one can take the definition directly to perform induction proofs with an appropriate mechanical theorem prover. Since the data involved are elements of carriers of canonical term algebras every induction is well-founded (structural induction). For example, one semantical property of a spec object of ALG is the consistency of the algorithmic definitions with the properties. They can be checked by structural induction proofs of every property by using the algorithmic definitions. (Note, that termination has to be considered separately.)

TR is based on this fact, and it tries to express SHEQ as ASPIK equations; the occuring function symbols are then defined algorithmically in some specification to be constructed (see sec. 5.1.3.). Therefore the ModPascal portions of MHEQ have to be eliminated and substituted by TR.

The rep-function (= carrier mapping) calls in HEQ are treated more tricky: since the rep-function definition is a mixture of ModPascal and ASPIK, an analogous elimination and substitution of ModPascal parts is performed in the definition. That yields to a pure ASPIK operation body, and calls of an algorithmi-

cally defined ASPIK operation. With this detour the occurences of rep-functions in MHEQ are integrated in an ASPIK formalism (we have, in fact, created a new rep-function by this process which we will call lifting; therefore a new rep-function identifier will be used in SHEQ). Note that lifting has to be semantics-preserving; see the formal definition 5.1.1.2.-2 below.

But how to eliminate the ModPascal parts of properties and carrier mapping definition? Otherwise AS would not be well-defined!

The idea is to exchange ModPascal constructs by 'semantical equivalent' ASPIK constructs to get a pure ASPIK specification AS to which the method is applicable. Since every ModPascal operation is associated to a defining module (or enrichment), the semantical equivalence is primarily defined on objects. It is advantageous to base this notion on realizations.

5.1.1.2.-1 Def. [semantical equivalent]
Let S ∈ Spec, M ∈ (MOD ∪ ENR).
Then S and M are called semantically equivalent , if there exists R ∈ Repobj with
(1) (S,M,R) is realization
(2) The signature morphism of R is bijective

■

Notation: SE(S, M) stands for 'S is semantically equivalent to M'.
SE(S, M, R) additionally selects a rep-object.

The new requirement imposed on realizations ensures the exchangability of ModPascal operations by ASPIK operations. Injectivity is not sufficient since it would be unclear how to treat additional module operations that occur in the MHEQ or in the carrier mapping definition.

A first approximation of TR for semantical equivalent objects is:

There are three actions:
A) replace a occurrence of a ModPascal operation identifier by the uniquely associated ASPIK identifier
B) replace the occurrence of a ModPascal variable as follows: since the variable is of a fixed type, and since this type is uniquely associated to an ASPIK sort, a new variable of that sort has to be generated and substituted in place of the ModPascal variable.
C) replace the occurence of a rep-function identifier as follows: apply steps A and B to the rep-function definition; establish the result (the 'lifted' rep-function) as a new operation identifier; substitute this identifier for the occurrence of the rep-function identifier.

If TR is applied to a set of syntactic structures, the steps A, B, C are performed.

¤

We write TR(S) for the result of the application of TR to a structure S.

With this definition of TR the lifting of a repfunction can be declared. To guarantee applicability of step c) we assume a set of primitive rep-functions with appropriate lifted versions.

5.1.1.2.-2 Def. [lifting]
Let S and M be semantically equivalent with R.
Let rf denote the rep-function of R.
Then the lifting $rf_{\llcorner}$ of rf is defined by $rf_{\llcorner} := TR(rf)$.

¤

For a given MHEQ, TR(MHEQ) yields in a tuple consisting of
- a modified MHEQ, that is nearly pure ASPIK; we denote this set by 'MHEQ'.
- lifted versions of the rep-functions.

The main point is that TR is only applicable for semantical equivalent objects i.e. if SE(S, M) holds; but S and M of a given re-co (S,M,R) do not yet have that property and therefore step A above will only be applicable for the definition of some used rep-function (which has occurrences of operations of semantically equivalent objects), but it does not modify MHEQ!

Therefore we introduce an intermediate step that will overcome this problem with ModPascal constructs in MHEQ (note that we we need to consider only this case; in the case of the rep-function definition of R it is enforced by syntax and semantics of rep-objects that only constructs of objects occur which are used by M (the module/enrichment involved in the connecting clause of the rep-object). In other words: no public operations of M occur in the rep-function definition.) The case of MHEQ is slightly different. Every equation embodies explicitly a public operation of M, and no operation of used objects. In this case a substitution-like exchange of ModPascal by ASPIK is impossible and senseless, since the validity of the equation is used to imply just this interchangability.

A solution is offered by the following considerations: in the homomorphy equation case, we are finally interested in the effect of a module operation call, where effect means either the induced state change or the represented value. To determine the effect we could use our denotational semantics function for ModPascal: we could compute $M[\![opcall]\!]\varsigma\sigma$ and substitute the result in MHEQ for opcall. But this is slightly orthogonal to our above described intention:

- $M[\![opcall]\!]\varsigma\sigma$ is based on an algebra function defined by an appropriate ModPascal operation definition opdef and $M[\![opdef]\!]$. Unfortunately, if we would substitute opcall by

M⟦opcall⟧σ in MHEQ, we would exchange ModPascal by a pure mathematical formalism, and getting no step closer to our goal of one formalism! Beside that we would torpedo our decision for an ASPIK formalism.

- If we would not only substitute but simultaneously evaluate the M terms, we might be more lucky. But evaluation of denotational semantics clauses involves sophisticated, not available machinery (c.f. the remark on this topic in sec. 5.1.1.1.). As a result we would not increase the degree of mechanization in our SEE.

We propose another way apart of denotational semantics terms that uses a symbolic evaluation step (sec. 5.1.1.2.1.), but is not free of problems (sec. 5.1.1.2.2.).

5.1.1.2.1. Symbolic Evaluation

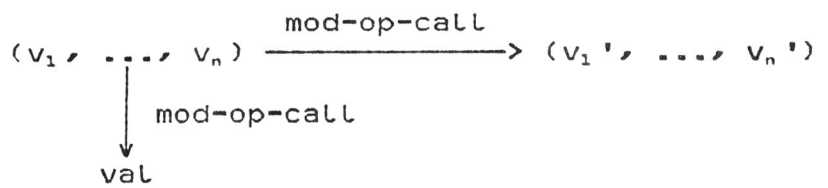
In this section we outline how symbolic evaluation can be used to achieve a symbolic representation of the effect of a ModPascal operation call.

Consider the design of module types in ModPascal. A mandatory component of these constructs are local variables. Every set of global variables of a module operation definition has to be a subset of this local variable set.

Let L_i, $i \in (n)$ denote the types of the local variables of a module M, and $(v_1, \ldots, v_n) \in (L_1 \times \ldots \times L_n)$ a vector of values of local variables. Every $(v_1, \ldots, v_n)$ is also called an internal state of the module incarnation (since this is just the information that the denotational semantics of ModPascal assigns to a variable of type M).

Then every module operation call either
- selects information from an internal state, or
- modifies an internal state.

or pictorially:



where val denotes an expression over $(v_1, \ldots, v_n)$, and the v_i', $i \in (n)$ are values of local variables after execution the operation call.

Now the main point is that in special cases the v_i' and val can be computed by symbolic evaluation in a way that no public operations of M occurs - due to the semantical restrictions imposed on module operation definitions. Then we can substitute ModPascal operation calls by equivalent (vectors of) expressions of the local variables of M which do not contain

occurrences of any public operation of M. Together with some hierarchy assumptions of sec. 5.1.2. we then are able to mechanically generate single formalism HEQ and to derive proof tasks suitable for mechanical first order theorem provers (see sec. 5.1.3. below).

Therefore we define a new action for our transformation algorithm and call the resulting set of actions TR-SYM:

TR-SYM	(algorithm for generation of SHEQ by symbolic evaluation)

TR-SYM consists of four actions:
A) replace an occurrence of a ModPascal operation identifier by the uniquely associated ASPIK identifier
B) replace the occurrence of a ModPascal variable as follows: since the variable is of a fixed type, and since this type is uniquely associated to an ASPIK sort, a new variable of that sort has to be generated and substituted in place of the ModPascal variable.
C) replace the occurrence of a rep-function identifier as follows: apply steps A and B to the rep-function definition; establish the result (the 'lifted' rep-function) as a new operation identifier; substitute this identifier for the occurrence of the rep-function identifier.
S) evaluate every call of a public operation of M symbolically by producing a vector of expressions of the local variables of M that represents the effect of the operation call; substitute the vector for the operation call.

The application of TR-SYM to a syntactic structure (MHEQ or rep-function definition) means:
1) perform S
2) perform A, B and C

The result of an application of TR-SYM to a set MHEQ is a tuple (SHEQ, rf_L) with
- SHEQ: the modified set MHEQ; formalism is (pure) ASPIK
- rf_L: a lifted version of rep-functions.

(see also the appendix for an illustrative example.)

An important condition for the soundness of this algorithm is that the symbolic evaluation coincides with our denotational semantics for ModPascal. But again: this is a once-and-never-again task which has to be performed by the designer of a SEE; we skip it here.

5.1.1.2.2. Current Limitations

The usability of symbolic evaluation to express the effect of ModPascal constructs in MHEQ depends on the limitations that come with this technique. Beside numerous technical problems ranging from integration of such a system into the context of

ASPIK and ModPascal up to administration of traversals, unfoldings/foldings of definitions etc. and preserving of consistency of the evaluation process, there are also severe theoretical difficulties.

For example, iterative structures cause problems since the number of repetitions is unknown in general. Then two solutions are thinkable: either
- counters are introduced that bound the traversals through a loop and that allow to relate variable values of different traversals, or
- loop invariants are introduced; then problems arise well-known from classical Hoare verification: How to get invariants? Are they strong enough? etc., and one looses much of the benefits of algebraic verification concepts.

Since both alternatives have far-reaching consequences on the proof method we do not involve one of them in the current paper and restrict ourselves to what is possible in our current framework. If situations with iterative structures occur, we will not apply symbolic evaluation but present an equation like (**) in sec. 5.1.1.1. to the user. Then he has interactively decide the validity of the equation, and then the system carries on with his answer (see the appendix for an example). With this limitations it is obvious that the class of operations and modules suitable for symbolic execution is not large enough to be successfully employed in practical experiences, and essential extensions are necessary. Nevertheless, in the (unexpected frequent) cases where our technique is nevertheless applicable, it mechanizes completely the generation of prooftasks suitable for automatic theorem provers and in the consequence the check of correctness of a realization, a fact that is highly valuable for the acceptance and performance of a software development system dedicated to verification issues.

5.1.2. Involvation of Hierarchy Information

Up to now a basic property of all components of a re-co (S,M,R) has not been considered: each object is hierarchical in that sense that it is based on already defined objects. Now the idea is to make assumptions concerning the hierarchies that allow us to consider only the top-elements in our correctness checks. This would free us from the necessity of resolving all use-relations before making correctness checks; the then generated three 'overall' objects would be of enormous complexity in general and not very well suited to mechanical treatment. (Note that (a) use-relations are of pure syntactical nature (structuring), (b) hierarchies are cyclefree, therefore resolution is possible, and (c) the validity of formulae is not affected by merging / separating formulae sets.)

The first assumption deals with predefined structures of ModPascal (types, type generators).
Let T := {BOOLEAN, INTEGER, CHAR, STRING} and

TC := {array, record, file, set, <enumeration>, <sub-range>}

(REAL omitted; the bracketed elements of TC denote the obvious type constructors.)

Now we assume that every element of (T ∪ TC) has a counterpart in ASPIK, i.e. that there is a set S(T) of algorithmic specifications and a set SC(TC) of algorithmic specification constructers such that ASPIK and ModPascal structures are uniquely associated.

The second assumption says that the associated objects are semantical equivalent, i.e. there are rep-objects for every ModPascal-ASPIK object pair such that 5.1.1.2.-1 is satisfied.

Both assumptions are easily satisfyable since they do not include user-defined objects; the installation of appropriate objects as system components is a solitary task.

The third assumption extends the first two to all objects (properly) used by any user-defined ModPascal object M of the re-co: every used structure of M' has an associated algorithmic ASPIK specification S', such that S' and M' are semantical equivalent.

The fourth and final assumption derived from the hierarchy property deals with the local objects of M. Every local object L is also assumed to be connected to a specification S_L such that the semantical equivalence holds. But since L is not explicitly used by M, we cannot consider the usual hierarchy relation. To model the situation we introduce a use-local relation that holds between a module and its local objects. The hierarchy notion for ModPascal objects is extended to allow both use- and use-local-relations. We use U(ob) to denote the used objects of ob, and Ul(ob) to denote the used local objects of ob.

To preserve consistency, an analogous modification is performed on the ASPIK level for those specifications that are semantical equivalent to a module. (Note that 'use-local' is equivalent to 'use' in the case of specification hierarchies).

Note that the last two assumptions can be reduced to the first two: every user-defined ModPascal object is built from elements of T ∪ TC; so by structural decomposition the assumptions on used and used-local objects may be reduced to the assumptions for T ∪ TC. Also, in the case of type generators the postulates above can be derived from the semantical equivalence of the base types. But there is no way to replace the assumptions for the latter.

Pictorially, we have for a re-co (S,M,R) the following hierarchies and assumptions:

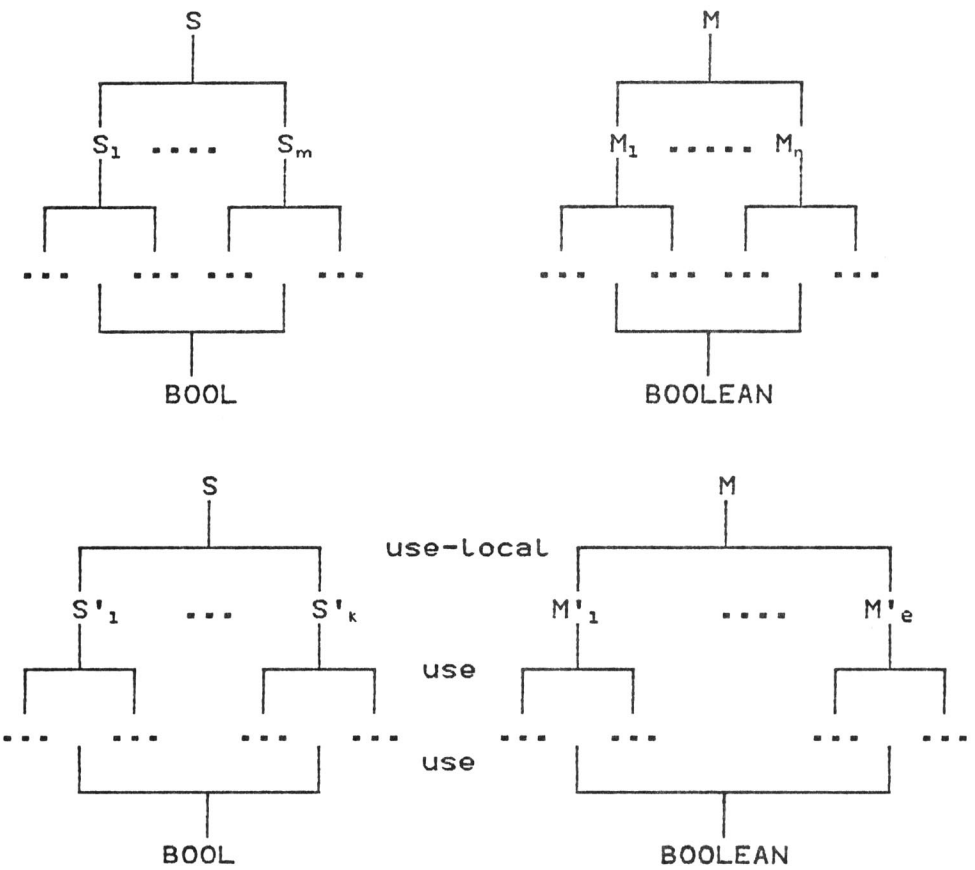


Fig. 5.1.2.-1: Object hierarchies

For the proof that S ∈ Spec and M ∈(MOD ∪ ENR) are semantically equivalent we proceed as follows:
(a) Supply R ∈ Repobj
(b) Show, that (S,M,R) is realization
(c) Show the bijectivity of the signature morphism of R

Let R ∈ Repobj be given.
To decide (b), we construct an inductive proof of the consistency of the artificial specification AS. The induction is leaned to the hierarchical structures that are induced by S and M via their use-relations.

The assumptions in this situation are:
- Every M_i is semantical equivalent to some S_j, i ∈ (n), j ∈ (m)
- Every M'_i is semantical equivalent to some S'_j, i ∈ (l), j ∈ (K)
- All other elements in the M hierarchies are semantical equivalent to some objects of the associated S hierarchies.

Note that these assumptions alone do not imply isomorphic

hierarchies.

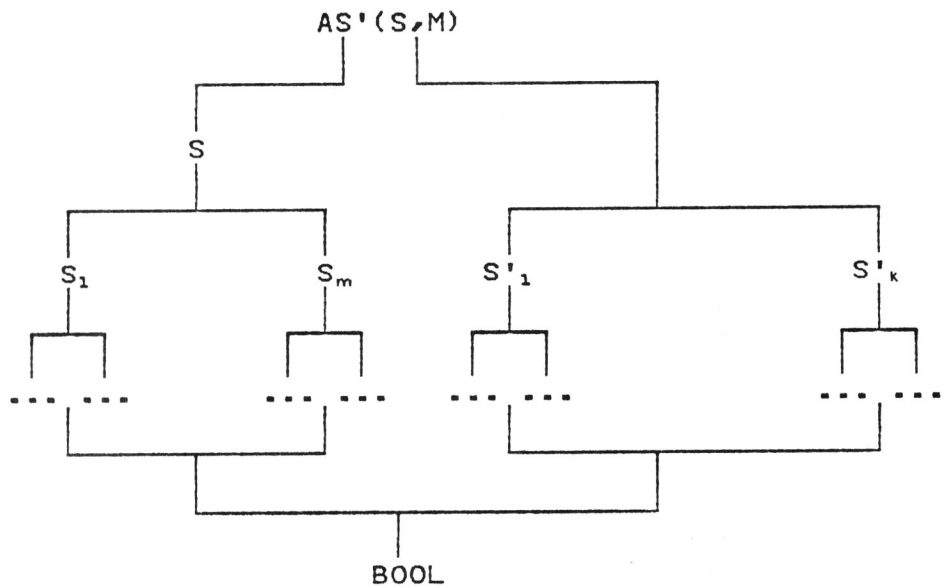
5.1.3. Formulation of the Induction Proof Task

As pointed out we will use our modified set SHEQ and the lifted rep-function rf_L to construct an artificial specification object AS'(S,M) for given re-co (S,M,R) with

- properties: SHEQ
- operation: rf_L

AS'(S,M) is well-defined: because all used ModPascal objects M_i are semantically equivalent to ASPIK objects S_i, we can exchange occurrences of operations of M_i by operations of S_i without causing harm. Also, module variables may be transformed in spec variables. And third, for all local objects M'_i of M there are semantically equivalent specs. In the case, when symbolic execution is applicable this can be used to remove from MHEQ the expressions that were substituted for the public operation occurrences. The expressions over local variables that were generated by symbolic execution of operation definitions in order to catch the effect of an operation call are transformable into ASPIK expressions by exchanging semantically equivalent operations and variables. From this it follows: both operations and properties of AS contain no ModPascal construct (i.e. AS $\in$ Spec).

Graphically we can construct AS'(S,M) with the notions of the previous section:



5.1.3.-1 Fig.: AS'(S,M) hierarchy

Since we want to have rf_L as single function of AS'(S,M) and rf_L is defined on S'_i, $i \in (K)$, then the above objects are

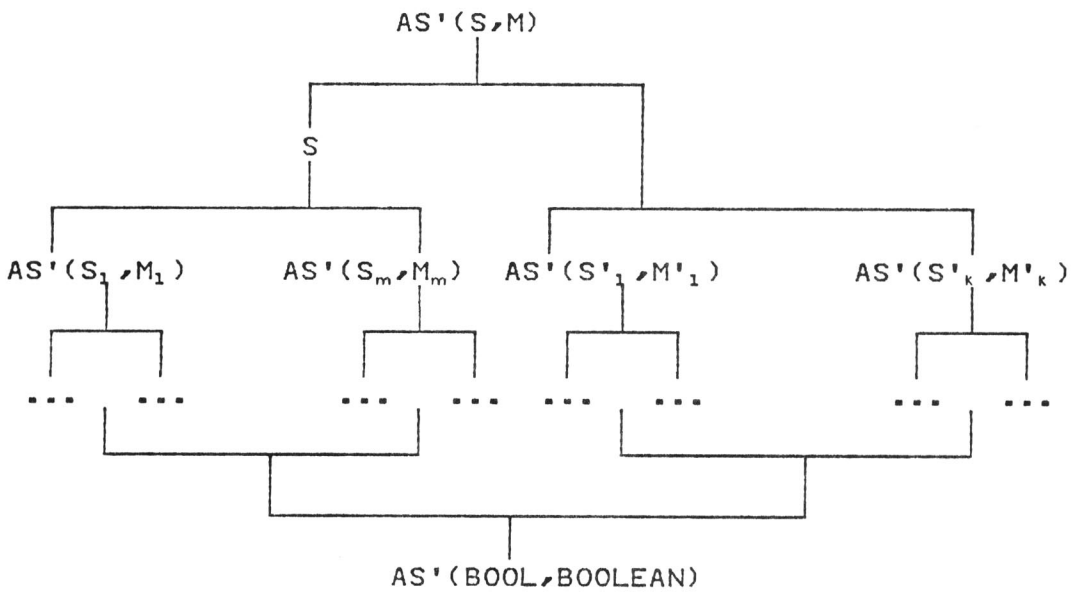
necessary. If sorts are ambiguously denoted by object names, then the functionality of rf_L is

$$S'_1 \times \ldots \times S'_k \Rightarrow S$$

Now the properties of AS'(S,M) (=SHEQ) just say that rf_L is a homomorphism. If we can show their validity we are nearly ready: we only have to derive from the homomorphy of rf_L the homomorphy of rf to satisfy our definition of realization. But this is a trivial step: since we generated rf_L from rf by substituting semantical equivalent operations, data and variables, every proposition for rf_L holds directly for rf. Therefore it is sufficient to show homomorphy for rf_L.

Up to now AS'(S,M) contains only the definition of rf_L but no definition of used rep-functions (which might occur in SHEQ as well as in the rf and rf_L definition). To include all potential necessary definitions we construct a hierarchy of AS-objects.

This is always possible if we involve the hierarchy information of S and M, according to the previous section. Especially, we assume that we have semantical equivalent hierarchies for S and M i.e. SE(U(S), U(M)) and SE(Ul(S), Ul(M)) hold. Under this assumption for every $S' \in (U(S) \cup Ul(S))$ and $M' \in (U(M) \cup Ul(M))$ the object AS'(S', M') is constructable. Since AS'-objects form also a hierarchy (starting with AS'(BOOL, BOOLEAN)) we have the following situation:



5.1.3.-2 Fig: AS-objects (isomorphic hierarchies for S,M)

We will use the following

Notation: AS(S, M) denotes an artificial specification constructed as AS'(S, M), but U(AS(S, M)) contains only S and other AS-objects.

Note that AS(S, M) contains definitions of all (lifted) rep-functions; it therefore will be used in proof scenarios.

To show properties of an ASPIK specification valid, there are two alternatives:
- deduce them from valid (other) properties
- show them consistent with algorithmic definitions of the occuring operations.

Deduction as well as consistency check involve support from mechanical theorem provers. So both possibilities are of similar complexity. But since AS(S,M) is algorithmic we take the consistency check to prove SHEQ.
In order to show consistency of an algorihmic specification one has to proceed along the following algorithm CON:

```
CON input: algorithmic, specterm-free,
           zero/one-sorted specification S
     output: true, if algorithmic definitions satisfy
                   the properties P(S) of S
             false otherwise

(1) if S contains no properties, then CON(S):=true
(2) for every property p of P(S).
    for all variables v_i of sort S_i occuring in P,
    i ∈ {1...u}.
    let ec_ij denote the set of elementary (=no arguments)
        constructors of sort S_i,
        i ∈ (n), j ∈ (q_i), q_i ∈ ℕ in
    let c_ij denote the non-elementary constructors
        of sort S_i, i ∈ (n), j ∈ (r_i), r_i ∈ ℕ in
    (2.1) let EC := {(ec_1j1,...,ec_njn)
                      | j_k ∈ (q_k), K ∈ (n)}
          denote the set of vectors of elementary
          constructors in
            for every (e_1,...,e_n) ∈ EC.
            p[(v_1,...,v_n)↤(e_1,...,e_n)] holds
    (2.2) let C := {(C_1j(1),...,C_nj(n))
                    | j_k ∈ (r_k), K ∈ (n)}
            denote the set of vectors of constructors in
          let ct_i denote a constructer term of sort S_i,
              i ∈ (n) in
          for every (e_1,...,e_n) ∈ C.
              if p[(v_1,...,v_n)↤(ct_1,...,ct_n)] holds
               then p[(v_1,...,v_n)↤(e_1<ct_1,...,ct_n,
                       ct'_1,...,ct'_a>,
                            e_n<ct_1,...,ct_n,ct'_1,...,ct'_a>)]
                    holds
                  where e_j<ct_1,...,ct_n,ct'_1,...,ct'_a>
                    denotes the constructor term build
                    from e_j, ct_i(1),...ct_i(b),ct'_j(1),...,
                    ct'_j(c), according to the arity
                    of e_j, with
                      ct'_i constructor term of some sort,
                      i ∈ (a), a ∈ ℕ
```

$\{i_1,\ldots,i_b\} \subseteq \{1,\ldots,n\}$,
$\{j_1,\ldots,j_c\} \subseteq \{1,\ldots,a\}$

(3) If both (2.1) and (2.2) hold, then CON(S) := true;
otherwise CON(S) := false

Remarks: a) The steps (2.1) and (2.2) together form a structural induction scheme that can be directly used by the proof system.

b) (2.1) can be shown by simple application of operation definitions (e.g. by use of an interpreter for ASPIK). Since EC is finite, a possible way is to check mechanically every alternative.

c) (2.2) represents the induction step: since the ct_i sets are countable (but well-founded), the only way to show the implication is to use an induction proof system that may take lemmata etc. from unfolding of operation definitions or from property sets of 'lower', already consistent specs.

■

In the case of AS(S,M) a second inductive scheme is implicitly employed as a result of our assumptions. The used and used-local objects M' of M are assumed to be semantically equivalent to some specifications S'. In this view we have a hierarchy of artificial specifications (starting with AS(BOOL,BOOLEAN)) that are all assumed to be consistent except

- the top-element AS(S,M)
- S
- used objects S' of S such that no used object M' of M is semantically equivalent to S'.

The consistency of S is not derivable from a semantically equivalence, since the latter is just the goal of these considerations! If consistency of S is needed it has to be shown explicitly. The same holds for objects S' above.
If additional requirements are imposed on the hierarchies ("isomorphic structure"), then the third kind of unknown consistency above will not occur. If in this case the consistency of S is shown separately, all properties of all used specifications of AS(S,M) may be employed in step (2.2) of CON, applied to AS(S,M).

It should be mentioned that the construction of artificial specifications AS(S,M) for M being ModPascal predefined type is not unproblematic. This is due to the fact that (1) the model of ModPascal standard types has to resolve cycles (see sec. 4 of [Olt 84a]), and (2) rep-functions are generally based on local variables; in the case of standard types there are no such variables!

The first point is mainly of technical nature: if the correct hierarchy of ModPascal objects is used, only the degree of complexity will increase.

The second problem can only be solved in a special treatment of rep-objects and rep-functions for ModPascal standard types. For example one could allow a missing rep-function definition in that case, and if carrier elements of the ModPascal structure occur one immediately switches over to the associated ASPIK structure and its carrier elements.
Independent of the choosen solution the necessary consequence is that employed algorithms and used proof systems have to be advised to handle correctly the standard object situation. Though we are conscious about the technical and theoretical problems arising we do not go into further detail and postpone a more comprehensive discussion.

We summarize the induction proof task:
Given the situation of figure 5.1.3.-2:
(a) Require isomorphic hierarchies for S and M
(b) Show CON(S) = true
(c) Show CON(AS(S,M)) = true
=> rf_L is homomorphism

5.1.3.-3 Lemma
Let S ∈ Spec, M ∈ (MOD ∪ ENR), R ∈ Repobj.
Let AS(S,M) be as above such that AS(S,M) ∈ Spec.
Then it is equivalent:
(S,M,R) is realization ⟺ AS(S,M) is consistent

¤

Remarks: a) AS(S,M) ∈ Spec implies pure ASPIK properties (and therefore previous applicability of symbolic execution).
b) The equivalence is exploited to check re-co's. The consistency of AS(S,M) may be shown by standard methods employed for non-artificial ASPIK specs, different from CON.

¤

5.1.4. Transfer of Proof Tasks

Once proof tasks suitable for mechanical theorem provers have been generated, a transfer to some available proof system has to be initiated. Since in general provers are designed to support one specific proof type (induction, rewriting, equality reasoning), it should be clear from the proof task which system has to be used.

Since we are interested in consistency proofs by induction (c.f. sec. 5.1.3.), one could automatically transfer generated proof tasks. Independently form the target system one will have to transform the ASPIK equations into the accepted input language. To get reliable results it has to be guaranteed that the transformation of proof tasks is semantically correct i.e. one has to perform another once-and-never-again task consisting in the check of 'semantical equivalence' of proof task representation. We assume that this has been done for our scenario.

There is no scheme for deciding, which equations beside those of AS(S,M) (e.g. all equations of the hierarchy?) should be physically attached to the prooftask. There are cases in which nearly every equation is necessary for a successful proof; sometimes the presence of redundant equations dramatically decreases efficiency or even makes a proof impossible. We assume an appropriate solution of this problem.

5.1.5. Administration

In SEEs, every software development will come with considerable number of proof tasks; for example, if specification hierarchies are realized in ModPascal then for every object there is a separate proof task. There has to be a satisfactory solution to the representation problem of proof tasks and to the administration of already achieved intermediate results. In particular:

- proof tasks should become objects by their own, with relations to data (specifications, modules) and tools.
- proof tasks contain their current state (proved or not).
- valid formulae are marked if a proof system or the software engineer validated them.
- pending proof tasks induce a lock on involved objects that hinders destructive access.
- there are inductive schemes which allow to incorporate already valid formulae into a proof task conveniently.
- there is a propagation algorithm that updates validity of proof tasks if destructive actions (as editing of correct objects) have occured.

All these features have to be tighly coupled to the user interface to allow efficient processing of proof tasks. We assume a SEE with comparable capabilities.

5.2. PMR

We now put together the single subtasks described in sec. 5.1. The result is an algorithm that guides the software engineer and the SEE in order to show the realization property. We do not explicitly distinguish whether single steps are performed manually or mechanically; this aspect was covered in the previous section.

The method for the proof of the realization property (PMR) then is defined as follows:

PMR
Let (S, M, R) denote a re-co. Let SE(U(S), U(M), U(R)) and SE(UL(S), UL(M), UL(R)) hold. (1) Let SM denote the signature morphism induced by R. If SM is not bijective, stop with failure. (2) Generate MHEQ

```
(3) Check if symbolic evaluation is applicable.
    If not, branch to (6).
    Otherwise: SHEQ := TR_SYM (MHEQ)↓1
               rf_L := TR_SYM (MHEQ)↓2
(4) Generate AS(S, M)
(5) Check, if CON(AS(S, M)) holds.
    If it is the case: stop with success
    Otherwise:         stop with failure
(6) Compute MHEQ with semantical operators.
    Look for external decision about the validity of
    MHEQ elements.
    Branch to (4)
```

Remarks a) The used and used-local objects are assumed to be already semantical equivalent.

b) The bijectivity of SM is not necessary if S and M are the final objects of the software development. In the other case this condition ensures that objects using M can be treated with PMR (see a). The main point is that for every concrete operation there has to be an abstract counterpart in order to perform technical steps as lifting or SHEQ generation. Together with the signature morphism property this induces bijectivity.

c) Note that after step 6 it is in fact possible to generate AS(S, M), since bijectivity of SM is assumed. The set MHEQ is modified by substitution of ModPascal constructs by ASPIK constructs, w.r.t. SM.

d) Stop with success means: (S, M, R) is a realization, or SE(S, M) holds. The negation "stop with failure" does not point to a unique source that causes the non-provability: either
- SM associates objects/operations wrong or inadequately, or
- S was inadequately or wrongly defined, or
- M does not what is specifies in S.

A thorough analysis of all possibilities has to follow.

This algorithm gives a rough overview on PMR. The details are skipped (e.g. how to treat 'mixed' cases, when for some operations of a module symbolic evaluation is possible, but not for others).

A system where PMR is implemented allows the interactive check of conditions that imply the correctness of a realization. Moreover, in some cases it is possible to mechanize the proof completely. This fact contributes enormously to the acceptance and applicability of the software development system since no specialists are needed to verify proof tasks.
It should be recalled that the embedding of PMR (as every verification method) requires a comprehensive and sophisticated object administration system that generates, inspects, manipulates, or propagates semantic properties of the

kind "is-realization". Consistency issues have to be solved arising from destructive operations as e.g. editing or erasing of objects.

6. Summary

This paper presents an overview on a solution of the implementation verification task arising in multi-level and multi-language software development environments. The situation is considered, when

- algebraic specifications for the abstract description, and
- module constructs for the concrete description

of non-concurrent behaviour are used. For both description levels the exemplary languages ASPIK and ModPascal are formally introduced. ASPIK is an algebraic specification language supporting hierarchical design of software; it provides verifiable notions for inter-object relations as 'refinement' or 'implementation', and offers a flexible object parameterization concept. ModPascal extends standard Pascal by a module construct and a type parameterization concept based on signature morphisms.

To connect a module M and a specification S the concept of representation object (rep-object) is introduced and supplied with a formal semantics. Rep-objects allow the user to define

a) a signature morphism between the specification and the module, and
b) a carrier mapping between the semantical algebras of the two objects.

The most important point is that - in contrast to 'abstract' approaches of e.g. [EKP 78], [EL 80], or [SW 82] - rep-objects model a relation between objects of different language levels (applicative and procedural). There are numerous difficulties induced by such a scenario, and to get started a satisfactory solution was found only by introduction of confinements: S is a single-sorted, specterm-free constructive spec (hierarchy); M contains no instantiate type definitions; i.e. not arbitrary ModPascal or ASPIK objects are considered in the relation induced by rep-objects. The notion of realization context rules out unadmissable objects.

If one can specify a representation object R that links S and M by a syntactical and a semantical mapping such that a homomorphism between the semantical algebras of S and M is induced, then a correct realization of S through M is achieved.

Under specific conditions the proof of the homomorphism property is mechanizable so that tedious and expert-dependent formal derivations are reduced. We present a comprehensive method for the treatment of these cases that also exploits the hierarchical structure of specifications and modules for inductive argumentation. The method is demonstrated by an

elementary example.

The concepts and the proof method have already been successfully employed in the ISDV-System where the realization check is only one of several verification tasks in order to determine consistency of requirement specification and imperative program. There, the mechanical tools for proving properties are an automatic theorem prover and a rewrite rule laboratory.

Within the ISDV system, several case studies on real world problems were successfully launched e.g. a financial accounting problem ([Olt 85b]).

Current and future research includes:

- Relaxation of the requirements 'zero/one - sortness' and 'constructivity' imposed on specification objects.
  As inevitable consequences the module concept of ModPascal has to be modified, and carrier mappings defined by rep-objects have to be considered between sets of algebras (instead of two constructively defined algebras).

- Involvation of spec-terms and instantiations / instantiate types in realization contexts.
  This seems to be a more technical issue because the constructs denote semantically ordinary specifications resp. modules/enrichments, such that the base case is applicable.

- Feasibility study if these concepts are adaptable to the development of software for non-sequential systems.
  In fact there is an ESPRIT project (GRASPIN) that is partially dedicated to a solution of this problem, and there is a close collaboration between the author and the GRASPIN team on this topic.

7. References

The following abbreviations are used:

CACM	Communications of the Association for Computing Machinery
DoD	Department of Defense
IJCAI	International Joint Conference of Artificial Intelligence
LNCS	Lecture Notes on Computer Science
SIGPLAN	Special Interest Group on Programming Languages

[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.

[ADJ 79] Thatcher, J.W., Wagner, E.G., Wright, J.B.: Data Type Specification: Parameterization and the Power of Specification Techniques. IBM Res. Rpt. RC 7757, TJW Res. Center, Yorktown, 1979.

[Bac 78] Backus, J.: Can Programming be liberated from the von-Neuman style? A functional style and its algebra of programs. CACM, 21, 8, 1978.

[Bak 80] Bakker, J. de: Mathematical Theory of Program Correctness. Prentice Hall, London, 1980.

[Bau 81] Bauer, F.L. et al.: Report on a Wide-Spectrum Language for Program Specification and Development. TU Munich, Report TUM-I8104, 1981.

[BG 77] Burstall, R. M., Goguen, J. A.: Putting Theories Together to Make Specifications. Proc. 5th IJCAI, pp.1045 - 1058, 1977.

[BR 85] Breiling, M., Rainau, U.: An Object Administration System and a Representation Object Programming System. Master thesis (in German). University of Kaiserslautern, 1985.

[BV 83] Beierle, C., Voss, A.: Parameterization - by - use for Hierarchically Structured Objects. University of Kaiserslautern, Memo SEKI-83-08, 1983.

[BV 85] Beierle, C., Voss, A.: Algebraic Specifications in an Integrated Software Development and Verification System. University of Kaiserslautern,

1985.

[Dij 74] Dijkstra, E.W.: A Simple Axiomatic Basis for Programming Language Constructs. Indagationes Mathematicae, 36 (1974), 1-15.

[Don 77] Donahue, J.: On the Semantics of "Data Type". Technical Report TR 77-311, Cornell University, 1977.

[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.

[EL 80] Ehrich, H., Lipek, U.: Algebraic Domain Equations. University of Dortmund, Report 125, 1981.

[Flo 67] Floyd, R.W.: AssigningMeanings to Programs. In: J.T. Schwartz (ed.): Proc. Symposium on Applied Mathematics, AMS, 19-37, 1967.

[Gan 82] Ganzinger, H.: Denotational Semantics for Languages with Modules. TU Muenchen, Inst. fuer Informatik 1982.

[GHM 79] Gordon, M., Milner, R., Wadsworth, C.: Edinburgh LCF. Springer, 1979.

[HKR 80] Hupbach, U. L., Kaphengst, H., Reichel, H.: Initial Algebraic Specifications of Abstract Data Types, Parameterized Data Types and Algorithms. VEB Robotron, Zentrum fuer Forschung und Technik, Dresden, 1980.

[Hoa 69] Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. CACM, 12, 576-580, 1969.

[Kla 82] Klaeren, H.: A Constructive Method for Abstract Algebraic Software Specification. Schriften zur Informatik und angewandten Mathematik, Bericht Nr.78, RWTH Aachen, 1982. [Lau 80]
Laut, A.: Safe Procedural Implementations of Algebraic Types. Information Processing Letters, pp. 147-151, 1980.

[Lic 85] Lichter, H.: An Interactive ans Syntax-oriented Inputsystem for Algebraic and Algorithmic Specifications. Master Thesis (in German), University of Kaiserslautern, 1985.

[Ma 74] Manna, Z.: Mathematical Theory of Computation. MacGraw-Hill, New York, 1974.

[Olt 84a] Olthoff, W.: ModPascal Report. University of

Kaiserslautern, Memo SEKI-84-09, 1984.

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

[Olt 85] Olthoff, W.: An Overview on ModPascal. SIGPLAN Notices, Vol. 20 (10) pp. 60 -71, 1985.

[Olt 85a] Olthoff, W.: The Realization Level. Internal Report. University of Kaiserslautern, 1985.

[Olt 85b] Olthoff, W.: Specification and Verification of a Real-World Book-Keeping Problem with SPESY: A Case Study. Internal Report. University of Kaiserslautern, 1985

[Pai 80] Pair C.: Abstract Data Types and Algebraic Semantics of Programming Languages. Report 80 - R - 011, Centre de Recherche en Informatique Nancy.

[Poi 83] Poigne, A., Voss, J.: Programs over Algebraic Specifications. On the Implementation of Abstract Data Types. Report 171, University of Dortmund, 1983.

[RL 85] Breiling, M., Eckl, G., Olthoff, W., Rainau, U., Schmitt, M., Weiss, P.: The RL-Handbook. Internal Report. University of Kaiserslautern, 1985.

[Roe 76] Roever, W.P. de: Recursive Program Schemes: Semantics and Proof Theory. Mathematisch Centrum, Amsterdam, 1976.

[Sch 82] Schulz, H.: Eine Methode zur korrekten Implementierung von Datentypen durch Module. Diplomarbeit, Uni Bonn 1982.

[Sch 85] Schmitt, M.: Extension of the ModPascal Precompiler (in German). University of Kaiserslautern, 1985.

[Spa 85] Spang, H.: Implementation of a Component of SPESY. Working paper (in German), University of Kaiserslautern, 1985.

[STA 79] Stanford Verification Group: Stanford Pascal Verifier User Manual. Comp. Science Dept. Stanford University, STAN-CS-79-731, 1979.

[SW 82] Sanella, D., Wirsing, M.: Implementation of Parameterized Specifications. Proc. 9th ICALP 1982, LNCS Vol. 140, pp 473 - 488, 1982.

[Weg 72] Wegner, P.: The Vienna Definition Language. Computing Surveys, Vol.4, No. 1, March 1972.

Appendix

This appendix tries to serve two purposes:
- illustration of the objects (specifications, modules, rep-objects) that are elements of the languages and of the scenario we have investigated; we present examples;
- illustration of concepts and algorithms of our approach in a 'micro' software development project with demands of verification.

Since this paper is intended to describe the theoretical foundations of the approach, the chosen example is rather tiny and simple. A closer-to-reality application can be found in [Olt 85b].

Our example is the famous stack, and we give definitions of it in ASPIK and ModPascal. In order to verify a transition from the ASPIK stack to the ModPascal stack, we define a rep-object and check, if the realization property holds.

A stack may be specified in ASPIK as follows:

```
spec STACK
use ELEM, NAT
sorts stack
ops push: stack elem ---> stack, pop: stack ---> stack,
    top: stack ---> elem, empty: ---> stack
props all s: stack all e: elem
       [P1]  pop(push(s,e)) == s
       [P2]  top(push(s,e)) == e
       [P3]  top(empty) == error
       [P4]  pop(empty) == error
```

```
spec-body
constructors empty, push
auxiliaries size: stack ---> nat
define-auxiliaries size(s) := case s is
                                *empty = 0
                                *push(s1,e1) = size (s1)+1
                                esac
define-carriers
     is-stack(s) := case s is
                    *empty: true
                    *push(s1,e1): if size(s1) < 10
                                  then true else false
                    esac
define-constructors
     empty := *empty
     push(s,e) := if size(s) < 10 then *push(s,e)
                                  else error
define-ops
     pop(s) := case s is *empty: error
                         *push(s1,e1): s1 esac
     top(s) := case s is *empty: error
                         *push(s1,e1): e1 esac
```

```
endspec
```

This specification exhibits most of the syntax of specs. Starred items denote carrier elements; ELEM, NAT and BOOL are assumed as already defined.

It should be recalled that the semantics of the axiomatic part of STACK (the spec header) consists of the category of unbounded stack algebras, whereas the algorithmic part (the spec body) restricts the semantics to the category of stack algebras 'of at most size 10'. As a whole, STACK possesses the second meaning.

A stack may be defined in ModPascal as follows:

```
type MSTACK =
     module use MELEM, MINTEGER;
       public procedure mpush(e: MELEM);
              procedure mpop;
              function mtop: MELEM;
              initial mempty;
       local type A = array[1:10] of MELEM;
             var a:A, i:MINTEGER; localend;
       procedure mpush;
              begin if i < 10
                     then begin i := i+1; a[i] := e end
                     else error                      end
       procedure mpop;
              begin if i=0 then error
                     else i := i-1                   end
       function mtop;
              begin if i=0 then error else mtop := a[i]
              end
       initial mempty;
              begin i := 0 end;
```

This definition shows a ModPascal version of bounded stack. MELEM and MINTEGER are assumed as already defined. Public operation arities omit a first parameter of type MSTACK; this parameter is supplied by the special syntax of module operation calls.

The algebra carrier introduced by MSTACK is the cartesian product (A x MINTEGER) i.e. tuples of array-integer values. The semantical operations behind mpush, mpop, etc. take these tuples as arguments and yield new tuples or select components.

We now want to specify a connection of both objects that

- maps the sort stack of STACK to the cartesian product sort of MSTACK and the STACK operations to their obvious conterparts in MSTACK (signature morphism)
- maps array-integer tuples (a,i) to STACK terms such that only significant values are associated to non-erroneous

terms (carrier mapping)

This is achieved by the rep-object RSTACK:

```
rep RSTACK
connecting STACK, MSTACK;
use RELEM;
ops  push = mpush
     pop = mpop
     top = mtop
     empty = mempty
repfct RSTACK(a,i) =
          if i=0 then empty else
          if 1≤i≤10 then push (RSTACK(a,i-1),
                               RELEM(a[i]))
             else error.stack
repend
```

Remarks: a) RELEM is an already defined rep-object for the obvious connection (although rep-object names are arbitrary in general).
b) The repfct RSTACK is ambiguously denoted by the rep-objects name.
c) The conditions of the if-clauses are pure ModPascal, the then- or else-branches are either pure ASPIK expressions or structures mixed of ASPIK portions, ModPascal portions, and recursive RSTACK calls.

¤

We now apply PMR to check if the re-co (STACK, MSTACK, RSTACK) is a realization.

(1) The signature morphism is bijective; the sort mapping is implicitly contained in RSTACK (stack ⇒ cartesian product type).
(2) The homomorphy equations MHEQ are:

[H1]	RSTACK(M.mpush(E))	=	push(RSTACK(M), RELEM))
[H2]	RSTACK(M.mpop)	=	pop(RSTACK(M))
[H3]	RSTACK(mempty)	=	empty
[H4]	RELEM(M.mtop)	=	top(RSTACK(M))

The dot notation in e.g. M.push(E) is equivalent to mpush(M,E). The variables M and E range over the concrete (cartesian product) carriers of MSTACK and MELEM.
(3) Symbolic execution is applicable to all equations of MHEQ. The local variable types of STACK are ARRAY and INTEGER. The internal state of a stack incarnation is therefore represented by a vector (a,i) where a and i are the local variables of STACK. We get the following formulae and expressions as effect of the operations:
- Symbolic representation of a stack object before ex-

ecution of a procedure : (a,i)
- Symbolic representation of a stack component object before execution of a function : (c)

Then we have
for mpush:
[S1] (i<10) ⇒ ({a,i,e},i+1)
[S2] ¬(i<10) ⇒ (undef,undef)

for mpop:
[S3] (i=0) ⇒ (undef,undef)
[S4] ¬(i=0) ⇒ (a,i-1)

for mempty:
[S5] (a,0)

for mtop:
[S6] ¬(i=0) ⇒ (a[i])
[S7] (i=0) ⇒ (undef)

Notational remark: {a,i,e} denotes the array a after assigning e to the i-th component. 'Undef' and 'undef' - vectors are the symbolic representation of erroneous evaluations.

The application of TR-SYM yield a set SHEQ and a lifted rep-function $RSTACK_{\llcorner}$.

We compute SHEQ in two steps:
- substitution of the results of the symbolic evaluation (SHEQ-1)
- substitution of renaming ModPascal constructs (built from used or local objects of STACK) and of RSTACK occurrences (SHEQ-2).

SHEQ-1:
The substitution is done in [H1] - [H4] for the operation calls mpush, mpop, mtop and mempty, according to [S1] - [S7].

[A1] (i<10) ⇒
RSTACK (({a,i,E}, i+1)) =
push(RSTACK((a,i)), RELEM(E))
[A2] ¬(i<10) ⇒
RSTACK((undef,undef)) = push(RSTACK((a,i)),RELEM(E))
[A3] (i=0) ⇒
RSTACK((undef,undef)) = pop(RSTACK((a,i)))
[A4] ¬(i=0) ⇒
RSTACK((a,i-1)) = pop(RSTACK((a,i)))
[A5] RSTACK((a,0)) = empty
[A6] ¬(i=0) ⇒
RELEM((a[i])) = top(RSTACK((a,i)))
[A7] (i=0) ⇒
RELEM((undef)) = top(RSTACK((a,i)))

RSTACK $_{\llcorner}$:
The reformulation of RSTACK is based on semantically equivalent specs for the ModPascal array and INTEGER

types. We assume the obvious specs with the obvious operations.

$RSTACK_L$: array x integer ⇒ stack

where array, integer and stack denote carriers of CTAs;

```
RSTACK_L(a,i) := if equal.integer(i,zero) then empty else
                 if between (succ(zero), i, succ10(zero))
                    then push(RSTACK_L(a,minus(i,1)),
                              RELEM_L(read(a,i)))
                    else error.stack
```

Note, that $RSTACK_L$ is a pure ASPIK operation.

SHEQ-2:
We modify [A1] - [A7] by replacing remaining ModPascal through ASPIK and carrier mapping calls through their reformulated version (RSTACK', RELEM').

```
[B1] less(i,succ10(zero)) ⇒
     RSTACK'(assign(a,i,e), plus(i,1)) =
            push(RSTACK'(a,i), RELEM'(e))
[B2] not(less(i, succ10(zero))) ⇒
     RSTACK'(error.array, error.integer) =
            push(RSTACK'(a,i), RELEM'(e))
[B3] equal.integer(i,zero) ⇒
     RSTACK'(error.array, error.integer) =
            pop(RSTACK'(a,i))
[B4] not(equal.integer(i,zero)) ⇒
     RSTACK'(a, minus(i,1)) = pop(RSTACK'(a,i))
[B5] RSTACK'(a,zero) = empty
[B6] not(equal.integer(i,zero)) ⇒
     RELEM'(read(a,i)) = top(RSTACK'(a,i))
[B7] equal.integer(i,zero) ⇒
     RELEM'(error.elem) ⇒ top(RSTACK'(a,i))
```

(4) The artificial spec AS(STACK,MSTACK) is:

```
spec AS(STACK,MSTACK)
use STACK, AS(ARRAY,MARRAY), AS(INTEGER,MINTEGER)
ops RSTACK_L: array x integer ⇒ stack
props /* [B1] - [B7] */
spec-body
define-ops RSTACK_L(a,i) := if equal.integer(i,zero)
                              then empty else
                            if between(succ(zero),
                                       i,succ10(zero))
                              then
                                push(RSTACK_L(a,minus(i,1)),
                                     RELEM_L(read(a,i)))
                              else error.stack
specend
```

Remark: The AS-objects are necessary since they contain

definitions of $RELEM_L$ and $RARRAY_L$. We omit details here.

(5) The object AS(STACK,MSTACK) has to be submitted to an automatic theorem prover. There, the consistency of [B1] to [B7] with the algorithmic definitions has to be checked by induction proof.
In our case we yield the result: (S,M,R) is realization, and the success is propagated in the SEE according to sec. 5.1.5!

We shortly scetch a situation, in which we currently branch to (6) in PMR (i.e. symbolic execution is not applicable):

(6) Assume a specification QUEUE of queues. It encloses an operation dequeue that removes the front element from a queue. QUEUE is also programmed as module MQUEUE with (among others) operation mdequeue. MQUEUE is represented analogously to MSTACK by array-integer tuples. Inserting an element in the queue is done by assigning it to an unused array element and increasing of the integer pointer. Mdequeue shifts the whole array one step left and decreases the integer pointer:

```
mdequeue(q) := if i=0 then error else
                  begin j: INTEGER; j := 1;
                        while j<i do begin
                            a[j] = a[j+1];
                            j := j+1; end
                        i := i-1 end
```

The occurrence of the while construct influences the applicability of symbolic execution; we then generate the equation

$$M[\![RQUEUE(Q.mdequeue)]\!]\varsigma\sigma \equiv M[\![dequeue(RQUEUE)]\!]\varsigma\sigma$$

and ask the user for the validity of this homomorphy equation; his answer is processed as if a proof system would have been used (i.e. the actions of sec. 5.1.5. are performed).