

A RULE-BASED APPROACH TO MODULAR SYSTEM DESIGN

Francesco Parisi-Presicce
Dipartimento di Matematica Pura ed Applicata
Universita' degli Studi - L'Aquila
I-67100 L'Aquila (Italy)

ABSTRACT

The problem of designing a modular software system which realizes a given specification starting from a predefined set of abstract data types is reduced to the problem of deriving the goal specification using the productions and then translating the derivation sequence (if it exists) into an interconnection of algebraic module specifications. The approach is based on the existence of a library of reusable modules, each represented by its visible interfaces in the form of productions.

1. INTRODUCTION

The development of correct programs is very difficult (if at all possible) without an appropriate formal support. Such a support can be provided by the algebraic approach with simple semantics and the central role played in algebraic specifications by the concept of abstract data type and the support this concept gives to the decomposition of large systems ([5]). For basic data types, a simple algebraic specification with sorts and operations with their equational properties is sufficient; the choice of a unique (up to isomorphism) algebra allows to abstract with respect to the data representation. In order to abstract from part of a specification to make it "variable" (parameter), it is necessary to use parametrized specifications along with the appropriate mechanisms for parameter passing [10]. For large software systems, it is useful to have another form of abstraction in order to hide internal details and to specify only the properties necessary for the interface between the user and the implementor. A formal concept of modules with an export interface was first proposed in [15]. The addition of an import interface and a parameter part leads to our notion of module specification ([10], [22], [1]). In our framework, a module specification consists of four parts: an import interface specifying what the module "needs" to operate, an export interface specifying the data type "produced" and made available to the outside by the module, a parameter part shared by the interfaces, and a body part in which the sorts and operations of the export interface are implemented in terms of those of the import interface. The items in the body and not in the export are hidden. The behaviour (or semantics) of a module is a transformation from data types which satisfy the import specification to data types which satisfy the export specification.

In order to develop large software systems in a stepwise manner, a flexible system of interconnections is needed to form the horizontal structuring of the system. The main interconnections viewed as operations on modules are: union, where the corresponding parts of two modules are put together by specifying the common subpart to be identified; composition, where the export interface of a module is matched with the import interface of another module;

actualization, where the parameter part is replaced by an actual specification. These (and other [17], [18]) mechanisms guarantee the "correctness" of the result given the correctness of the arguments and the compositionality of the semantics ([1], [11]).

There is an analogy between this notion of module and that of MODULA-2 and ADA. A MODULA-2 module is subdivided into a "definition module", the visible part, and an "implementation module", corresponding roughly to our export interface and body part. The interconnection mechanism in MODULA-2 is "strict": the union is realized implicitly (syntactically) while for the composition, modules can import other items, not by specifying the properties that they must satisfy but by referencing explicitly other modules. More details in [14]. A similar comparison with ADA has been reported in [1], where packages correspond to modules with an empty parameter and generics to the more general case. In both cases, only syntactic aspects, i.e., types and operations, of a component specification are allowed, leaving it to comments in a natural language to convey their meaning to user and implementor (with the obvious danger of incompleteness and misinterpretation). In our algebraic approach, semantical aspects are included in the specification of the interfaces. In the simplest framework of the basic algebraic case, the four parts are represented by algebraic specifications, with equations or positive conditional equations as axioms, and are related by specification morphisms. The formalism has been extended to include algebraic constraints and first order logical formulas ([8]) or specifications in more general institutions.

The import and export interfaces, along with the shared parameter, are the only part visible to the outside and provide the gates through which each module interacts with the other ones. The basic idea proposed is to use the interfaces as productions or rules and to apply them to a specification. Given a library of (correct) modules represented by their interfaces as productions of specifications, the problem of developing the horizontal structure for a modular system to realize a GOAL specification given a set of predefined data types is viewed as the problem of symbolically deriving (a specification containing) the GOAL specification from the specification PRE of the predefined types using the given productions. To each derivation and each combination of derivations corresponds an operation on the module specifications which realize the productions used. The sequence of derivations from PRE to GOAL can be translated into a modular design whose behavior is to transform any model of PRE into a model (realization) of the objective GOAL. The discussion here is restricted to the derivations and the translations from the derivations to module design. We postpone to future papers other important aspects such as

strategies of search or choice of applicable productions based on problem specific knowledge.

R.M.Burstall and J.A.Goguen :

Another important factor for the practical utilization of abstract specification languages is to build up a library of specifications which can then be used in putting together other large specifications. Without such a library, every program specification effort will have to start from scratch and there will be no significant progress...

2. MODULES AND PRODUCTIONS

In this section, we briefly review some basic notions of algebraic specifications [10] and the concept of module specification ([1]) with the operations used for the horizontal structuring mechanism of module specifications. For ease of presentation, we consider only the basic algebraic case of module specifications without constraints (in the sense of [8]). By algebraic specification SPEC we mean a triple (S,OP,E) consisting of a set S of sorts, a set OP of operation symbols and a set E of (positive conditional) equations. To distinguish among the parts of different specifications, we subscript the components, i.e. refer for example to the second component of the algebraic specification SPEC1 as OPSPEC1. Sig(SPEC) denotes the signature (S,OP) of SPEC and, for $N \in OP$, sorts(N) denotes the set of sorts of the operator symbol N. Eqn(Sig(SPEC)) is the set of equations over (S, OP) and a set X of variables. A specification morphism $f: SPEC1 \rightarrow SPEC2$ is a signature morphism $(f_S, f_{OP}) : (S1, OP1) \rightarrow (S2, OP2)$ which associates in consistent way operator symbols to operator symbols and sorts to sorts and such that the translation $f^\#(E1)$ of the equations of SPEC1 is contained in E2 (it can be extended [19] to $f^\#(E1)$ derivable from E2 or to equationally closed sets E of equations). Each specification morphism $f: SPEC1 \rightarrow SPEC2$ defines a forgetful functor $V_f: Alg(SPEC2) \rightarrow Alg(SPEC1)$ (which intuitively translates models of SPEC2 into models of SPEC1 disregarding possible additions introduced by f) and a free functor $F_f: Alg(SPEC1) \rightarrow Alg(SPEC2)$ (which intuitively builds for each model A of SPEC1 the model of SPEC2 with the least amount of properties compatible with the specification SPEC2, using the elements of A as constants). In all our examples, we use the following more intuitive notation, listing under the key words sorts, opns and eqns the elements of the three components of the specification.

FS-PAR = bool +

sorts f*, dest, dep

opns EQF : f* f* \rightarrow bool

NODEP : \rightarrow dep

eqns EQF(F*, F*) = TRUE

denotes the specification obtained by adding to the standard specification of boolean values the set of sorts $S' = \{f^*, \text{dest}, \text{dep}\}$, the two operation symbols EQF and NODEP (the second one with an empty argument list of sorts denotes a constant) and the equation $EQF(F^*, F^*) = \text{TRUE}$ universally quantified over all the F^* ranging over elements of sort f^* . Considering the obvious inclusion of bool in FS-PAR as a specification morphism f, the forgetful functor V_f applied to any model of FS-PAR returns its boolean part ignoring all elements of sorts f^* , dest or dep.

2.1 Definition [13], [1]

A module specification MOD consists of four algebraic specifications PAR (parameter part), EXP (export interface), IMP (import interface) and BOD (body) and four specification morphisms as in the following commutative diagram

$$\begin{array}{ccc} & e & \\ \text{PAR} & \rightarrow & \text{EXP} \\ i \downarrow & s & \downarrow v \\ \text{IMP} & \rightarrow & \text{BOD} \end{array}$$

A module specification is correct if the free functor $F_S : Alg(IMP) \rightarrow Alg(BOD)$ satisfies $V_S(F_S(A)) = A$ for all IMP-algebras A. The semantics SEM of MOD is the functor $V_V F_S : Alg(IMP) \rightarrow Alg(EXP)$.

Interpretation We say that MOD realizes its interface Int(MOD) = (IMP \leftarrow PAR \rightarrow EXP) which represents the only information available outside the module. Both the export and the import interface are "contained" in the body specification, which provides an implementation of EXP by IMP. The operations and sorts in BOD but not in EXP are to be considered hidden. The semantics of MOD is a transformation between algebras : the import specifies the kind of algebra to be provided to the module to obtain an algebra which satisfies the export interface. The export and import interfaces share a parameter part, which, if the module is correct, is the only part of any IMP-algebra A guaranteed to be left unchanged by the semantical transformation. If PAR=IMP and EXP=BOD, this notion coincides with that of parametrized specification.

2.2 Example

The example of a module for a flight schedule consisting of triples with flight number (f*), destination and departure, is taken from [1]. It avoids duplication of flight numbers with the export equation on ADD-FS, which is implemented in the body by the hidden operation TAB which keeps an unordered list of triples. FS-MOD = (FS-PAR, FS-EXP, FS-IMP, FS-BOD) where FS-PAR is the specification given above, FS-IMP is exactly FS-PAR, while

sorts fs

opns CREATE-FS : \rightarrow fs

SEAR-FS : f* fs \rightarrow bool

ADD-FS : f* dest dep fs \rightarrow fs

RET-FS : f* fs \rightarrow dep

CHAN-FS : f* dep fs \rightarrow fs

eqns SEAR-FS(F*, CREATE-FS) = FALSE

SEAR-FS(F*, FS) = TRUE \Rightarrow

ADD-FS(F*, DEST, DEP, FS) = FS

SEAR-FS(F*, FS) = TRUE \Rightarrow

RET-FS(F*, CHANGIFS(F*, DEP, FS)) = DEP

and

FS-BOD = FS-EXP +

opns TAB : f* dest dep fs \rightarrow fs

eqns ADD-FS(F*, DEST, DEP, CREATE-FS) =

TAB(F*, DEST, DEP, CREATE-FS)

ADD-FS(F*1, DE1, DEP1, TAB(F*, DE, DEP, FS)) =

if EQF(F*1, F*) then TAB(F*, DE, DEP, FS)

else TAB(F*, DE, DEP, ADD-FS(F*1, DE1, DEP1, FS))

SEAR-FS(F*, CREATE-FS) = FALSE

SEAR-FS(F*, TAB(F*1, DEST, DEP, FS)) =

EQF(F*, F*1) or SEAR-FS(F*, FS)

```

RET-FS(F*,CREATE-FS) = NODEP
RET-FS(F*,TAB(F*1,DEST,DEP,FS)) = if EQF(F*,F*1)
    then DEP else RET-FS(F*,FS)
CHAN-FS(F*,DEP,CREATE-FS) = CREATE-FS
CHAN-FS(F*,DEP,TAB(F*1,DEST1,DEP1,FS)) =
if EQF(F*,F*1) then TAB(F*,DEST1,DEP,FS) else
    TAB(F*1,DEST1,DEP1,CHANFS(F*,DEP,FS)).

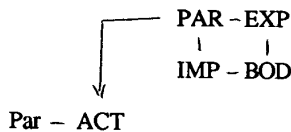
```

The four specification morphisms as in 2.1 are all inclusions. The operation TAB in the body is used to implement the exported operation ADD but is not visible to the outside. The semantics of the module reflects the fact that the operations in the body not already in the import have only the properties specified by their (equational) definitions and their consequences (in the usual equational calculus) of these definitions. In other words, the defining equations are to be seen as rewrite rules to compute the value of the functions. With this view of the body equations, the correctness of the module specification corresponds to requiring that the rewrite rules just mentioned define a total function (termination and confluency of the rules) if the result is of a sort of the import algebra. No such restrictions are imposed on operations with result value of a new sort.

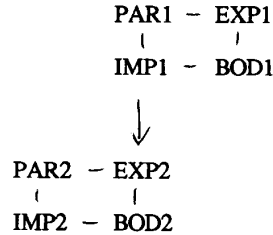
There are three basic operations which, when applied to module specifications and parametrized specifications, yield new module specifications. Such operations guarantee the correctness of the resulting module if the arguments are correct and allow to express the semantics of the result in terms of the semantics of the arguments of the operation. We describe these operations informally, referring the reader to [1], [11] and [14] for details and the formal definition.

The operation of union $MOD1 + MOD0 MOD2$ of two module specifications MOD1 and MOD2 with respect to a common submodule MOD0 is obtained by taking for each component of the result the disjoint union of the corresponding components of MOD1 and MOD2 and then identifying their common part of MOD0. Only MOD0 (possibly up to renaming) determines the parts not to be duplicated. The resulting module specification is automatically correct and its semantics is given by the "amalgamated sum" $SEM1 + SEM0 SEM2$ of the corresponding semantics ([2], [11]).

The operation of actualization $act_h(PS, MOD)$ consists of replacing the parameter part PAR of MOD by a (parametrized) specification $PS = (Par, ACT)$ via a parameter passing morphism $h: PAR \rightarrow ACT$ which associates the actual sorts and operations of ACT with the formal ones of PAR. The replacement of PAR by ACT is propagated to the interfaces and the body part (since PAR is "contained" in all three). Any new import algebra is composed of an ACT-algebra and an IMP-algebra: the semantics of the actualized module $act_h(PS, MOD)$ leaves unchanged the ACT-part and transforms the IMP-part according to SEM ([16],[11]).



The third basic operation on module specifications is that of composition $MOD1 \cdot MOD2$, where the import interface of a module specification is "matched" with the export interface of another one via a specification morphism h . The "unused" interfaces provide two of the components of the composite module specification and the "intersection" of the parameters is the new parameter. The new body is the body BOD1 where IMP1 has been replaced by BOD2. The sorts and the operations of EXP2 are no longer exported by the composite module, whose semantics is the composition of the original ones, interposed by the possible translation of the matched interfaces $(SEM1 \cdot V_{hE} \cdot SEM2)$. The resulting module is again correct if MOD1 and MOD2 are ([10],[11])

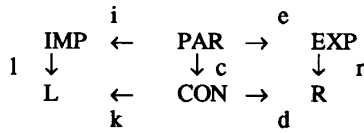


These basic operations, along with the other ones of product, partial composition, recursion (cyclic dependencies of modules) and iteration satisfy several algebraic laws which allow different but semantically equivalent strategies for interconnecting module specifications.

Each module consists of four parts, but only three of them, namely the two interfaces and the parameter part, are visible to the system designer or in general to the user of the module specification. The body contains the construction of the export interface operations in terms of those of the import interface and the internal representation of the exported data. Only the designer of the module has access to the body of the module. The interfaces (and their shared parameter part) provide the "gates" through which each module interacts with the other modules and contain the information about the module behavior which is visible to the user of the module. So there is a clear separation between what the module does (i.e. transforms IMP-algebras into EXP-algebras) and the details of how this transformation is implemented (in the body). The interfaces of a module specification then provide the information necessary to reason about its properties and behavior and that are relevant for the reuse of the module. Our approach to system design is based on viewing the interfaces of a module specification as a production and on applying these productions to specifications. If starting from an initial specification, the goal specification is generated using the productions, then the goal specification can be realized through a modular system built using the realizations of the productions (library modules) and the way in which the productions are applied (interconnection of modules).

2.3 Definition [19]

A SPEC-production is an ordered pair $PRO = (IMP \leftarrow PAR \rightarrow EXP)$ of injective specification morphisms $i: PAR \rightarrow IMP$ and $e: PAR \rightarrow EXP$. A direct derivation via the production PRO consists of the following two pushout diagrams of specifications



A production PRO is applicable to a specification L if there exist a morphism $l : IMP \rightarrow L$ and a context specification CON such that L is the pushout (gluing) of IMP and CON. The result R of the derivation is the pushout of EXP and CON. In this case we write $PRO : L \Rightarrow R$ and say that R is derivable from L via PRO.

Interpretation The middle specification PAR represents the part of IMP left unchanged by the production. The specification CON in the direct derivation is the part of L not affected by the derivation : it is the "context" of the derivation and it is "glued" to the right end side EXP of the production via the common subspecification PAR, which represents the "interface" between the unchanged context and the modified part (from IMP to EXP). Notice that a direct derivation is symmetric and that if $PRO : L \Rightarrow R$, then $PRO^{-1} : R \Rightarrow L$, where $PRO^{-1} = (EXP \leftarrow PAR \rightarrow IMP)$.

2.4 Example

The interfaces of the FS-MOD of Example 2.2 along with the obvious inclusions form a production of algebraic specifications FS-PRO. When this production is applied to the specification $L = nat + string$ where

```

nat = bool +
  sorts    nat
  opns     0 : → nat
           succ : nat → nat
           Eq : nat nat → bool
  eqns     Eq(succ(x), succ(y)) = Eq(x, y)
           Eq(x, x) = TRUE
           Eq(0, succ(x)) = FALSE
           Eq(succ(x), 0) = FALSE

and
string = sorts    alph, str
  opns     EMPTY : → str
           MAKE : alph → str
           CONC : str str → str
  eqns     CONC(CONC(x, y), z) = CONC(x, CONC(y, z))
           CONC(EMPTY, x) = x = CONC(x, EMPTY)

```

via the specification morphism $l : FS-IMP \rightarrow nat+string$ with $l_S(f^*) = nat = l_S(dep)$, $l_S(dest) = str$, $l_{OP}(EQF) = Eq$, $l_{OP}(NODEP) = EMPTY$, it produces the specification

```

R = L +
  sorts    fs
  opns     CREATE-FS : → fs
           SEAR-FS : nat fs → bool
           ADD-FS : nat str nat fs → fs
           RET-FS : nat fs → dep
           CHAN-FS : nat nat fs → fs
  eqns     SEAR-FS(N, CREATE-FS) = FALSE
           SEAR-FS(N, FS) = TRUE =>
           ADD-FS(N, ST1, ST2, FS) = FS

```

SEAR-FS(N, FS) = TRUE =>
RET-FS(N, CHAN-FS(N, ST1, FS)) = ST1

Notice that in this example the effect of applying the production is to add to the specifications of nat and string the five operator symbols labelled with -FS. This is the case when the production is deductive, i.e., when $PAR=IMP$. In this case, it is sufficient to have a specification morphism $l : IMP \rightarrow L$ to obtain a direct derivation via the production. In general, a production $PRO = (IMP \leftarrow PAR \rightarrow EXP)$ is applicable to the specification L if there exists a context specification CON and a specification morphism $c : PAR \rightarrow CON$ such that $L = IMP + PAR \cdot CON$, i.e., if L can be decomposed into two parts IMP and CON which share exactly the PAR part of the production. PAR represents the "boundary" between the part IMP involved in the derivation via PRO and the part CON not affected by the production. The result of the derivation is then the specification obtained by "gluing" CON and EXP via PAR. In general, the "pushout complement" $CON = L - PAR \cdot IMP$ need not exist.

Intuitively, the specification CON should contain all the sorts of PAR and the sorts of L not contained in IMP, i.e., $S_{CON} = S_L - l_S(S_{IMP} - i_S(S_{PAR}))$. Similarly for the operator symbols $OP_{CON} = O_{PL} - l_{OP}(O_{PIMP} - i_{OP}(O_{PPAR}))$. It is obvious that for CON to be well defined, the operator symbols cannot use sorts not contained in the set of sorts S_{CON} . Similarly, the equations $E_{CON} = E_L - l^{\#}(E_{IMP} - i^{\#}(E_{PAR}))$ of CON cannot be formed using operator symbols not in OP_{CON} if we want a properly defined specification. If the "occurrence" morphism $l : IMP \rightarrow L$ is injective, this is all that is needed for l to be applicable. If not, then there are some technical conditions that require the items collapsed by l to be "gluing" items.

2.5 Theorem

If $l : IMP \rightarrow L$ is a specification morphism for the production $PRO = (IMP \leftarrow PAR \rightarrow EXP)$ and $DANG = \{s \in S_{IMP} : N \in O_{PL} - l_{OP}(O_{PIMP}) \text{ and } l_S(s) \in \text{sorts}(N)\}$
 $ID_S = \{s \in S_{IMP} : s' \in S_{IMP}, s' \neq s, l_S(s) = l_S(s')\}$
 $ID_{OP} = \{N \in O_{PIMP} : N' \in O_{PIMP}, N \neq N', l_{OP}(N) = l_{OP}(N')\}$.
then the existence of CON such that $L = IMP + PAR \cdot CON$ is equivalent to the following three conditions
1) $i_S(S_{PAR}) \supset DANG \cup ID_S$
2) $i_{OP}(O_{PPAR}) \supset ID_{OP}$
3) $Eq_N((S_{CON}, OP_{CON})) \supset E_{CON}$

3. FROM DERIVATION TO DESIGN

Suppose now that we have been able to generate the goal specification from the initial one using as productions the interfaces of the library modules. How can we now use the way the goal is derived to design a modular system which realizes this goal, that is, which transforms a model of the initial specification into a model of the goal specification? The modular system is built using the definition of applicability of a production and the way parallel and concurrent derivations appear in the generation.

First notice that if R is derivable from L via the production $PRO = (IMP \leftarrow PAR \rightarrow EXP)$ then any L-algebra is the amalgamated sum ([1]) of an IMP-algebra I and a CON-algebra C via the common PAR-algebra P, written $I +_P C$.

CON specifies the part of L left unchanged by the derivation, either because in PAR or because outside the part affected by IMP. Therefore the derivability of R from L induces a semantical transformation from $I + p C$ to $SEM(I) + p C$, the only one compatible with the semantical transformation of PRO from I to $SEM(I)$. All this is just the general case of the (specific) interpretation of derivation as operation on module specification. Recall that MOD realizes PRO if $PRO = Int(MOD)$.

3.1 Theorem [19]

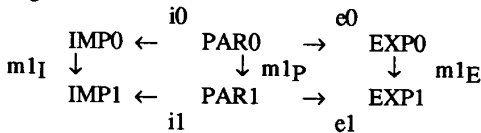
If R is derivable from L via the production $PRO = (IMP \leftarrow PAR \rightarrow EXP)$ as in 2.3 and MOD realizes PRO, then there exists a module specification MOD' transforming L-algebras into R-algebras. The module MOD' consists of the module MOD where the parameter PAR has been actualized by CON via $c : PAR \rightarrow CON$. Its semantics SEM' is given by $SEM'(I + p C) = SEM(I) + p C$.

Going back to Example 2.4, the derivability of R from L via the interfaces of FS-MOD implies the existence of a module $act(nat + string, FS-MOD)$ which has **nat** + **string** as import and as export interface the specification

$R = L +$ sorts fs
opns CREATE-FS : $\rightarrow fs$
 SEAR-FS : $nat\ fs \rightarrow bool$
 ADD-FS : $nat\ str\ nat\ fs \rightarrow fs$
 RET-FS : $nat\ fs \rightarrow dep$
 CHAN-FS : $nat\ nat\ fs \rightarrow fs$
eqns SEAR-FS(N, CREATE-FS) = FALSE
 SEAR-FS(N, FS) \Rightarrow ADD-FS(N, ST1, ST2, FS) = FS
 SEAR-FS(N, FS) = TRUE \Rightarrow
 RET-FS(N, CHAN-FS(N, ST1, FS)) = ST1

In order to translate a derivation sequence into system design, we must investigate the relationship between the interaction of two derivations and that of the two modules realizing the corresponding productions.

The simplest case is that of a derivation sequence $PRO1 : L \Rightarrow R$, $PRO2 : R \Rightarrow R'$ where the overlap of the two productions in R consists of common gluing elements only, i.e., when $r1(e1(PAR1)) \cap l2(i2(PAR2)) \supset r1(EXP1) \cap l2(IMP2)$. Such sequence is called sequentially independent. In this case, the production PRO2 does not need the elements added to R by the first production PRO1 and can therefore be applied directly to L to produce R". The two direct derivations $PRO1 : L \Rightarrow R$ and $PRO2 : L \Rightarrow R'$ so obtained are parallel independent, i.e., they too intersect only in common gluing elements and can therefore be applied in parallel (simultaneously). This discussion is summarized in the next theorem which needs the definition of subproduction. A production PRO0 is a subproduction of PRO1 if there exist specification morphisms as in the following commutative diagrams



3.2 Theorem (Parallelism)

Let $PRO1 : L \Rightarrow R$ and $PRO2 : R \Rightarrow R'$ be a sequentially independent derivation sequence. Then

- 1) PRO2 can be applied to L producing a specification R"
- 2) the derivations $PRO1 : L \Rightarrow R$ and $PRO2 : L \Rightarrow R'$ are parallel independent
- 3) for some subproduction PRO0 of PRO1 and PRO2 with $IMP0 = PAR0 = EXP0$, the parallel production $PRO1 + PRO0 PRO2$ is applicable to L and $PRO1 + PRO0 PRO2 : L \Rightarrow R'$.

Consider as production the following interfaces PS-PRO of a module specification which builds a plane schedule (ps) with no duplications of the plane number (p^*).

PS-IMP = PS-PAR = **bool** +
sorts p^* , type, seats
opns EQP : $p^* p^* \rightarrow bool$
eqns EQP(P^* , P^*) = TRUE

PS-EXP = PS-PAR +
sorts ps
opns CREATE-PS : $\rightarrow ps$
 SEAR-PS : $p^* ps \rightarrow bool$
 RESERVE-PS : $p^* type\ seats\ ps \rightarrow ps$
eqns SEAR-PS(P^* , CREATE-PS) = FALSE
 SEAR-PS(P^* , PS) = TRUE \Rightarrow
 RESERVE-PS(P^* , TYPE, SEATS, PS) = PS

As shown in Example 2.4, the production FS-PRO can be applied to $L = nat + string$ producing the specification R shown there. The production PS-PRO can now be applied to R via the specification morphism $l2 : PS-IMP \rightarrow R$ with which $l2_S(p^*) = l2_S(seats) = nat$, $l2_S(type) = str$ and $l2_{Op}(EQP) = Eq$.

Notice that since PS-PRO is deductive, there is no need to check the conditions of Theorem 2.5.

The application of PS-PRO to R produces the specification $R' = nat + bool +$

sorts fs, ps
opns CREATE-FS : $\rightarrow fs$
 SEAR-FS : $nat\ fs \rightarrow bool$
 ADD-FS : $nat\ str\ nat\ fs \rightarrow fs$
 RET-FS : $nat\ fs \rightarrow nat$
 CHAN-FS : $nat\ nat\ fs \rightarrow fs$
 CREATE-PS : $\rightarrow ps$
 SEAR-PS : $nat\ ps \rightarrow bool$
 RESERVE-FS : $nat\ str\ nat\ ps \rightarrow ps$
eqns SEAR-FS(N, CREATE-FS) = FALSE
 SEAR-FS(N, FS) = TRUE \Rightarrow
 ADD-FS(N, ST1, ST2, FS) = FS
 SEAR-FS(N, FS) = TRUE \Rightarrow
 RET-FS(N, CHAN-FS(N, ST, FS)) = ST
 SEAR-PS(N, CREATE-PS) = FALSE
 SEAR-PS(N1, PS) = TRUE \Rightarrow
 RESERVE-PS(N1, STR, N2, PS) = PS

The two derivations $FS-PRO : L \Rightarrow R$ and $PS-PRO : R \Rightarrow R'$ are sequentially independent because the images of FS-EXP and PS-IMP in R overlap only in **nat** + **string** which in turn is contained in the image of the gluing part (FS-PAR and PS-PAR) of each of the two productions. By Theorem 3.2 we can anticipate the application of PS-PRO directly to L using the same specification morphism l2 defined above. The subproduction PRO0 of (3) of the theorem above

is just $PRO0 = (bool \leftarrow bool \rightarrow bool)$ and the parallel production $FSPS-PRO = FS-PRO + PRO0$ $PS-PRO$ is given by
 $FSPS-PAR = FSPS-IMP = FS-IMP + bool PS-IMP$
 $FSPS-EXP = FS-EXP + bool PS-EXP$

By applying the parallel production $FSPS-PRO$ to L we obtain in one direct derivation the specification R' above.

Given that the original productions are the interfaces of module specifications from the library, we can use these modules to build (define) another one which realizes the parallel production $FSPS-PRO$. If we call by $Mbool$ the module specification where each part is the boolean specification $bool$ and each morphism the identity, then the parallel production $FSPS-PRO$ consists of the interface of the union $FS-MOD + Mbool PS-MOD$ of the module specifications $FS-MOD$ and $PS-MOD$ with respect to their shared (subparameter) part $Mbool$ ([16],[11]). Combining this with Theorem 3.1, we obtain a module specification $act_1(nat + string, FS-MOD + Mbool PS-MOD)$ whose import and export specifications are L and R' respectively. Note that we could have obtained the same result by first taking the disjoint union of the modules $FS-MOD$ and $PS-MOD$ and then actualizing the result by the actual specification $nat + string$: this actualization identifies the two copies of $bool$ coming from the two distinct modules. Similarly, we could have taken the disjoint union $FS-PRO + PS-PRO$ of the productions in the Parallelism Theorem: the subproduction of (3) of that theorem is not unique and can be chosen in a certain range (between the empty specification and exactly the overlap of the two derivations). The corresponding modules are different, but the subsequent actualized ones are the same.

The parallel production of Theorem 3.2(3) is a special case of a more general operation on productions: amalgamation. If we start with a subproduction $PRO0$ of $PRO1$ and $PRO2$ we can define the amalgamation of $PRO1$ and $PRO2$ w.r.t. $PRO0$ as the production $(IMP1 + IMP0 IMP2 \leftarrow PAR1 + PAR0 PAR2 \rightarrow EXP1 + EXP0 EXP2)$ denoted by $PRO1 + PRO0 PRO2$. The UNION Theorem of [19] states that if $PRO0$ is a sub-production of $PRO1$ and $PRO2$ and each production $PROj$ is applicable to a specification Lj to produce the specification Rj in a "consistent" way (that is the effect of $PRO1$ and $PRO2$ on the common subpart $L0$ is the same and exactly that of $PRO0$) then $PRO1 + PRO0 PRO2 : L1 + L0 L2 \Rightarrow R1 + R0 R2$ i.e., consistent productions on parts ($L1$ and $L2$) of the same specification ($L1 + L0 L2$) can be amalgamated into one production.

Another interaction between sequential derivations that can be translated into an interconnection of module specifications is matching. A pair of derivations $PRO1 : L1 \Rightarrow L2$ and $PRO2 : L2 \Rightarrow L3$ is matched if $EXP1 = IMP2$ and the morphisms $r1 : EXP1 \rightarrow L2$ and $l2 : IMP2 \rightarrow L2$ are the same. For such a sequence of matched derivations, we can construct a new production which can accomplish the same global transformation. The composition of two productions $PRO1 = (IMP1 \leftarrow PAR1 \rightarrow EXP1)$ and $PRO2 = (IMP2 \leftarrow PAR2 \rightarrow EXP2)$ is defined when $EXP1 = IMP2$, it is denoted by $PRO2 \circ PRO1$ and is given by $(IMP1 \leftarrow PAR3 \rightarrow EXP2)$ where $PAR3$ is the pullback (intersection) of $PAR1$ and $PAR2$ in $EXP1$. As might be expected, the result of applying the

composite production $PRO2 \circ PRO1$ is the same as the sequential application of the two productions (first $PRO1$ and then $PRO2$)

3.3 Theorem(Matched Concurrency)

Let $PRO1 : L1 \Rightarrow L2$ and $PRO2 : L2 \Rightarrow L3$ be a pair of matched derivations. Then the composition of $PRO1$ and $PRO2$ can be applied to $L1$ and $PRO2 \circ PRO1 : L1 \Rightarrow L3$.

To illustrate the result of the Theorem, suppose that we have another library module whose interface consists of

$APS-PAR = FS-PAR + bool PS-PAR$

$APS-IMP = FS-EXP + bool PS-EXP$

$APS-EXP = APS-PAR +$

sorts aps

opns $CREATE : \rightarrow aps$

$SEAR : f^* aps \rightarrow bool$

$RET : f^* aps \rightarrow dep$

$CHAN : f^* dep aps \rightarrow aps$

$SCHED : f^* dest dep p^* type seats aps \rightarrow aps$

eqns

$SEAR(F^*, CREATE) = FALSE$

$SEAR(F^*, APS) = TRUE \Rightarrow$

$SCHED(F^*, D, DEP, P^*, TYPE, SEATS, APS) = APS$

$SEAR(F^*, APS) = TRUE \Rightarrow$

$RET(F^*, CHAN(F^*, DEP, APS)) = DEP$

This production can be applied to the specification R' by the morphism $l2 : APS-IMP \rightarrow R'$ given by

$l2_S(f^*) = l2_S(p^*) = l2_S(dep) = l2_S(seats) = nat;$

$l2_S(dest) = l2_S(type) = str ; l2_S(fs) = fs ; etc.$

The result of such a direct derivation is the specification

$L3 = nat + string +$

sorts aps

opns $CREATE : \rightarrow aps$

$SEAR : nat aps \rightarrow bool$

$RET : nat aps \rightarrow nat$

$CHAN : nat nat aps \rightarrow aps$

$SCHED : nat str nat nat str nat aps \rightarrow aps$

eqns

$SEAR(N, CREATE) = FALSE$

$SEAR(N, APS) = TRUE \Rightarrow$

$SCHED(N1, ST1, N2, N3, ST2, N4, APS) = APS$

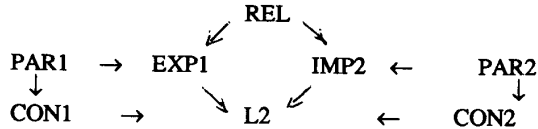
$SEAR(N, APS) = TRUE \Rightarrow$

$RET(N, CHAN(N, N', APS)) = N'$

The direct derivations $FSPS-PRO : nat + string \Rightarrow R'$ and $APS-PRO : R' \Rightarrow L3$ satisfy the hypothesis of the Concurrency Theorem. We can therefore define the composition $APS-PRO \circ FSPS-PRO$ of the productions to obtain the production $AFPS-PRO = (AFPS-IMP \leftarrow AFPS-PAR \rightarrow AFPS-EXP)$ where $AFPS-IMP = FSPS-IMP$, $AFPS-PAR = APS-PAR$ and $AFPS-EXP = APS-EXP$.

We can now apply the production $AFPS-PRO$ to the initial specification $nat + string$ to obtain in only one derivation the specification $L3$ above. The module specification corresponding to this derivation is the actualization by $nat + string$ of the composition of the $APS-MOD$ module which realizes the APS production above, with the union module $FS-MOD + Mbool PS-MOD$.

The Matched Concurrency Theorem deals only with the simplest case of interaction between two sequential derivations (besides of course the sequentially independent case). Consider two derivations $PRO1 : L1 \Rightarrow L2$ and $PRO2 : L2 \Rightarrow L3$ and a "relation" for them, i.e., a specification REL and two specification morphisms $b1 : REL \rightarrow EXP1$ and $b2 : REL \rightarrow IMP2$ such that in the diagram



the middle diamond commutes. The matched Concurrency Theorem deals with the case where $REL = EXP1 = IMP2$ and the two morphisms are the identity. There are more general cases where the two derivations can be combined into one using a composite production.

Two similar and symmetric cases are those where either $REL = EXP1$ or $REL = IMP2$. In the first case, $EXP1$ is only part of the specification $IMP2$ and therefore the part of $IMP2$ not provided for by $PRO1$ must be "pushed back" and joined with $IMP1$. The IMP -part of the composite production consists then of $IMP1$ and $IMP2 - b2(EXP1)$, its gluing part of the common part of $PAR1$ and $PAR2$ and its EXP -part consists of $EXP2$ only (since everything "generated" by $PRO1$ has been "used" by $PRO2$ via the morphism $b2$). The new production can be realized by taking the partial composition ([17]) of the realizations of the productions $PRO1$ and $PRO2$. The symmetric case where $REL = IMP2$ can be constructed in a similar way, "adding" for the final EXP -part the specification $EXP2$ and the part $EXP1 - b1(IMP2)$ of $EXP1$ not "used" by $PRO2$. The realization of this composite production can be accomplished by using the operation of partial product ([18]) on the single realizations.

The most general case of an arbitrary relation REL for a sequence of derivations has been considered in [12] and has required a composite interconnection mechanism to construct its realization from the realizations of the component productions.

4. ADAPTING THE DESIGN

It is quite unlikely that the library at our disposal contains all the productions needed to generate our goal specification. More likely is the situation where part of the goal specification can be realized using the given modules and part must be implemented anew. We illustrate the situation with an example. Call SPEC the following specification of a system to handle the activity of a library such as build and manipulate a list of books and a list of patrons, associate borrowed books with borrowers, keep track of the due dates for the RET of borrowed books, etc.

SPEC = bool +

sorts title, author, duedate, books,
patno (number identifying a patron),
type (of patron), phone, patrons, library
opns CREATE : \rightarrow books

LOOK : title books \rightarrow bool

RESERVED : title author duedate books \rightarrow books

DATE : title books \rightarrow duedate

UPDATE : title duedate books \rightarrow books

NODATE : \rightarrow duedate

EQT : title title \rightarrow bool

NEW : \rightarrow patrons

FIND : patno patrons \rightarrow bool

ADD : patno type phone patrons \rightarrow patrons

START : \rightarrow library

RETURNED : title library \rightarrow library

SEAR : title library \rightarrow bool

BORROW : title author duedate patno type phone library \rightarrow library

eqns SEAR(TI, START) = FALSE

SEAR(TI, LIB) = TRUE \Rightarrow

BORROW(TI, AUT, DD, PN, TYP, PHO, LIB) = LIB

LOOK(TI, BK) = TRUE \Rightarrow DATE(TI, UPDATE(TI, DD, BK)) = DD

LOOK(TI, BK) = TRUE \Rightarrow RESERVED(TI, AUT, DD, BK) = BK

LOOK(TI, CREATE) = FALSE

EQT(TI, TI) = TRUE

FIND(PN, NEW) = FALSE

FIND(PN, PAT) = TRUE \Rightarrow ADD(PN, TYP, PHO, PAT) = PAT

RETURNED(TI, START) = START

RETURNED(TI, BORROW(TIL, AUT, DD, PN, TYP, PHO, LIB))

= if EQT(TI, TIL) then LIB else

BORROW(TIL, AUT, DD, PN, TYP, PHO, RETURNED(TI, LIB))

We have listed the sorts by their intuitive name to simplify the reading. The elements of sort title, author and type are intended to be strings, while duedate, patno and phone are to be represented as natural numbers. The specification GOAL to be realized is SPEC with such a substitution.

Now we want to know *if* and *how* this specification can be realized using the library of modules represented by the productions FS-PRO, PS-PRO and APS-PRO illustrated in the previous sections and starting from the initial specification **nat + string**.

As we have seen in section 2, we can apply FS-PRO to **nat + string** to produce

SP1 = nat + string +

sorts fs

opns CREATE-FS : \rightarrow fs

SEAR-FS : nat fs \rightarrow bool

ADD-FS : nat str nat fs \rightarrow fs

RET-FS : nat fs \rightarrow dep

CHAN-FS : nat nat fs \rightarrow fs

eqns

SEAR-FS(N, CREATE-FS) = FALSE

SEAR-FS(N, FS) = TRUE \Rightarrow ADD-FS(N, ST1, ST2, FS) = FS

SEAR-FS(N, FS) = TRUE \Rightarrow

RET-FS(N, CHAN-FS(N, ST1, FS)) = ST1

To this specification we can apply, as illustrated in section 3, the production PS-PRO to obtain

SP2 = nat + bool +

sorts fs, ps

opns CREATE-FS : \rightarrow fs

SEAR-FS : nat fs \rightarrow bool

ADD-FS : nat str nat fs \rightarrow fs

RET-FS : nat fs \rightarrow nat

CHAN-FS : nat nat fs \rightarrow fs

CREATE-PS : \rightarrow ps

```

SEAR-PS : nat ps → bool
RESERVE-FS : nat str nat ps → ps

eqns
  SEAR-FS(N, CREATE-FS) = FALSE
  SEAR-FS(N,FS)=>
    ADD-FS(N, ST1, ST2, FS) = FS
  SEAR-FS(N,FS)=TRUE=>
    RET-FS(N, CHAN-FS(N,ST,FS))=ST
  SEAR-PS(N, CREATE-PS) = FALSE
  SEAR-PS(N1,PS)=>
    RESERVE-PS(N1,STR,N2,PS)= PS

```

Now we can apply the production APS-PRO to generate the specification

```

SP3 = nat + string +
  sorts    aps
  opns
    CREATE : → aps
    SEAR : nat aps → bool
    RET : nat aps → nat
    CHAN : nat nat aps → aps
    SCHED : nat str nat nat str nat aps → aps

eqns
  SEAR(N,CREATE) = FALSE
  SEAR(N,APS)=TRUE=>
  SCHED(N1,ST1,N2,N3,ST2,N4,APS)=APS
  SEAR(N,APS)=TRUE=> RET(N,CHAN(N,N',APS))=N'

```

Finally we can apply again FS-PRO and PS-PRO (in any order since they are sequentially independent) to SP3 to obtain the specification

```

SP4 = nat + string +
  sorts    aps
  opns
    CREATE : → aps
    SEAR : nat aps → bool
    RET : nat aps → nat
    CHAN : nat nat aps → aps
    SCHED : nat str nat nat str nat aps → aps
    CREATE-FS : → fs
    SEAR-FS : nat fs → bool
    ADD-FS : nat str nat fs → fs
    RET-FS : nat fs → nat
    CHAN-FS : nat nat fs → fs
    CREATE-PS : → ps
    SEAR-PS : nat ps → bool
    RESERVE-FS : nat str nat ps → ps

eqns
  SEAR-FS(N, CREATE-FS) = FALSE
  SEAR-FS(N,FS)=> ADD-FS(N,ST1,ST2,FS) = FS
  SEAR-FS(N,FS)=TRUE=>
    RET-FS(N, CHAN-FS(N,ST,FS))=ST
  SEAR-PS(N, CREATE-PS) = FALSE
  SEAR-PS(N1,PS)=>
    RESERVE-PS(N1,STR,N2,PS)= PS
  SEAR(N,CREATE) = FALSE
  SEAR(N,APS)=TRUE=>
  SCHED(N1,ST1,N2,N3,ST2,N4,APS)=APS
  SEAR(N,APS)=TRUE=> RET(N,CHAN(N,N',APS))=N'

```

Notice now that with a simple renaming (fs stands for books, ps for patrons and aps for library) which can be accomplished by a production $REN = (OLDNAMES \leftarrow OLDNAMES \rightarrow NEWNAMES)$

where the first morphism is the identity and the second one the one-to-one and onto mapping performing the renaming, we have our GOAL specification with the exception of the operator RETURNED. We have generated the specification SP4 while what is required is the specification $GOAL = SP4 +$

```

  opns RETURNED : str → aps
  eqns RETURNED(TI,START)=START
  RETURNED(TI,BORROW(TIL,AUT,DD,PN,TYP,PHO,LIB))
    = if EQT(TI,TIL) then LIB else
  BORROW(TIL,AUT,DD,PN,TYP,PHO,RETURNED(TI,LIB))

```

Using the Parallelism Theorem and the Matched Concurrency Theorem of the last section, we can define the modular system which realizes the specification SP4 by taking :

- *FIRST* the union $FS-MOD + Mbool PS-MOD = FSPS-MOD$ of the modules (realizing the productions FS-PRO and PS-PRO used in the first 2 steps) which when actualized by $nat + string$ would give SP2 as export interface;
- *THEN* the composition $AFPS-MOD = APS-MOD \bullet FSPS-MOD$ of the module obtained in the previous step and the module realizing the production APS-PRO (the specification SP3 can be obtained as export after actualizing this composite module by $nat + string$)
- *THEN*, with the Parallelism Theorem again, the union $FS-MOD + Mbool PS-MOD$ of the realizations of the productions FS-PRO and PS-PRO used to generate SP4 from SP3 and combine this module with the one used to generate SP3 by taking their union (one more application of the Parallelism Theorem)

We can now exploit the compatibility of the actualization of module specifications with other operations on modules ([16],[11]) and the connection, expressed in Theorem 3.1, between a derivation via a production and the actualization of the realization of the production. By actualizing the last module specification obtained by the parameter passing morphism $l : PAR \rightarrow nat + string$, we obtain the module $act_l(nat + string,$

$(APS-MOD \bullet FSPS-MOD) + Mbool FSPS-MOD)$

whose export interface contains the specification SP4.

We are in a much better position than we were at the beginning : instead of constructing a module specification to implement, starting from $nat+string$, fourteen operations satisfying the properties expressed in the specification SPEC, we only need to implement the operation called RETURNED, the only one not contained in our generated specification. The problem of obtaining the operation RETURNED can be solved in (at least) three different ways, all of which entail the addition, via *extension*, of the operation to the modular system.

- DIRECT EXTENSION

This approach is the most immediate one and it consists of adding the operation to the export interface and (hence) to the body part. At this point, an attempt is made at defining the operation RETURNED in terms of the other operations present in the body (which contains all the operations from import and export) and then to show that the properties which RETURNED must satisfy are a consequence of the equations satisfied by the other operators. Since new operations and

equations are added to the body of the module, the correctness of the new module must be proven. Unfortunately, the extension is not a "clean" operation (in the sense of [7]) in that it does not guarantee the correctness of the result based on the correctness of the parts. We must then prove the correctness property of 2.1 and the task is not trivial because it involves the body of the total module. To the difficulty of proving correctness, we must add the possibility that the new operation cannot be implemented in terms of the import operations. This situation would then require an extension of the import interface and, along with the proof of correctness as before, a revision of the applicability of the overall module to the specification of the predefined types. One more drawback of this approach is the loss of modularity of the system : the resulting module cannot in general be decomposed into the union or composition of self-contained modules and the new operation is reusable only to the extent that the whole module is . In our example there is no need to add anything to the import of the module. The GOAL specification can be realized as the export interface of $\text{act } 1 \text{ (nat+string, ext (AFPS-MOD +Mbool FSPS-MOD, \{RETURNED: \dots\}))}$

- PARTIAL EXTENSION

This approach consists of extending only part of the module specification constructed so far. The basic idea is that RETURNED is part of a subspecification SP of GOAL. Since only the operation RETURNED of GOAL has not been generated, there must be a point in the derivation where the subspecification SP - {RETURNED} is derived. It is at this point in the generation that we can extend the module by adding to it the missing operation. We search therefore for the "closest" derivation to SP and add RETURNED to the modular system designed up to that point. In our example, RETURNED is part of the subspecification containing START, BORROW and SEAR and therefore can already be added to the realization of SP3. The GOAL specification can be realized as the export interface of $\text{act } 1 \text{ (nat+string, ext (AFPS-MOD, \{RETURNED \dots\}) +Mbool FSPS-MOD)}$ As in the previous case, there is no need to add new items to the import interface to implement RETURNED but the (necessary) correctness proof is simpler than in the *DIRECT EXTENSION* because the module to be proven correct is smaller (AFPS-MOD instead of AFPS-MOD+Mbool FSPS-MOD). Furthermore, the system has maintained some of its modular structure, in particular all the modular structure deriving from subsequent applications of productions. Since the modules realizing these last productions are not affected by the extension, no new proof of their correctness is required.

-CLEAN EXTENSION

This is the most desirable way of adding the part of the GOAL specification not generated. It consists of designing a new module which implements RETURNED and the smallest part of SP necessary to have a well defined specification. This new module is then combined (using the union operation) with the module realizing the rest of SP (and with which it shares a common submodule). The new module, smaller in general than the one obtained with *PARTIAL EXTENSION*, must be proven correct and compatible, in its shared part, with the one already realizing the rest of SP. Besides the smaller correctness proof, this approach preserves the modular structure of the system since the missing items are added to

the rest using clean operations such as union and composition. In our particular example, this approach does not give the expected improvements over the *PARTIAL EXTENSION* approach. This is due to the fact that the equations characterizing RETURNED contain occurrences of the operations START, EQT and BORROW and the axiom of the last one contains SEAR as well: there is no meaningful decomposition of the specification that will lead to a nontrivial union of two modules. What can be done is to notice that, since the addition of RETURNED does not require an extension of the import of APS-MOD, we can restrict our extension to this last module and then compose it with the realization of the productions used earlier. Using this observation, we obtain GOAL as the export interface specification of the module

$\text{act } 1 \text{ (nat+string, (ext (APS-MOD, \{RETURNED \dots\}) \cdot \text{FSPS-MOD}) +Mbool FSPS-MOD)}$

5.CONCLUDING REMARKS

In this paper, we have proposed an approach to the development of software systems based on ideas and techniques from the algebraic theory of Graph Grammars ([6]). Given an initial specification, SPEC-productions are applied sequentially or in parallel to generate another specification : if the productions are the interfaces of module specifications from a library, then the derivation sequence can be translated into a modular system. The results obtained are intended as a formal support for a rule-based (expert) system to aid the systematic development of large software systems from a library of reusable components. We have discussed in details the notions of derivability and of translation of a derivation sequence into an interconnection of modules and we have seen how to anticipate the application of a production using the Parallelism Theorem. This equivalence of derivations translates into provably equivalent modular systems. There may be several derivation sequences which generate the GOAL specification and each may give a different system, with different characteristics such as size of modules, number of interconnections, etc. Even for a given sequence, there may be different systems realizing the GOAL : we are investigating ([12]) the manipulation of a given sequence into "canonical" form to obtain a canonical system with the smallest number of interconnections.

We have discussed how to apply a rule and how to translate the result, but not the strategies necessary in the choice of the rule. The development could be purely syntactic, trying all possible rules, or could be directed, possibly in an interactive way, by knowledge specific to the problem, giving priority to the rules based on semantical aspects not represented in the specification. Along the same lines, we have assumed the existence of an "occurrence" morphism $l : \text{IMP} \rightarrow L$ and then established the conditions for the applicability of the production. The choice of l could also be directed by other semantical conditions. Furthermore, the possibility may arise that a given morphism is consistent with respect to sorts and operations but not with respect to the axioms : at this point a design decision is needed on whether or not to add the property to what is being realized.

We have discussed the horizontal structure of the system and how to recover it from the derivation. What needs to be investigated also is how the derivation (and hence the system) must change to accommodate the "vertical" development ([7]) of

modules and system, with notions such as refinement and simulation. We have pointed out the symmetry of the direct derivation in 2.3. This symmetry allows us to perform a search of the GOAL specification in a generative or forward mode similar to the derivation of a string in a grammar or in a goal-oriented or backward mode starting from the GOAL and applying the productions from EXP trying to arrive at base cases of predefined types (more on this in a forthcoming paper). Also the derivation sequence can be used as documentation on the decomposition of the final system.

A module specification can be viewed as the description of an implementation: the body of the module provides an implementation of the sorts and operations of the export interface in terms of those of the import. The export specification is implemented (as in [20]) by the import specification via the constructor functor SEM : Alg(IMP) → Alg(EXP). The transformation induced by the semantics of a module specification can also be viewed as an implementation functor. In general our approach is useful for other notions of implementation, using transformations from Alg(IMP) to Alg(EXP) other than the semantics of modules, for the constructors in the sense of [20]. The development process using their notion of implementation corresponds to a sequence of direct derivations.

The objective of programming "in the small" is to produce a program (i.e., an element in a language generated by a grammar) which "solves" a problem specified in some formalism. Programming "in the large" is reduced to the same problem, as what is needed is to produce an element of a grammar, keeping track of the "production" process.

ACKNOWLEDGEMENT This research was supported in part by the Consiglio Nazionale delle Ricerche under "Progetto Finalizzato : Sistemi informatici e Calcolo Parallelo" and in part by the Ministero della Pubblica Istruzione

REFERENCES

- [1] E.K.Blum,H.Ehrig,F.Parisi-Presicce,"Algebraic Specification of Modules and their Basic Interconnections", *J. Comp. System Sci.*34, 2/3 (1987) 239-339
- [2] E.K.Blum,F.Parisi-Presicce,"The Semantics of Shared Submodule Specifications", Proc. CAAP85 /TAPSOFT85, Lect. Notes in Comp. Sci. 185, Springer-Verlag 1985, pp.359-373
- [3] R.M.Burstall,J.A.Goguen,"Putting Theories together to make Specifications", Proc.5th Int. Conf. on Artificial Intelligence (1977) 1045-1058
- [4] R.M.Burstall, J.A.Goguen, "An informal introduction to Specifications using Clear" in 'The correctness Problem in Computer Science' (R.S.Boyer, J.S.Moore eds.) Academic Press 1981
- [5] B.Krieg-Bruckner, ed., "A Comprehensive Algebraic Approach to System Specification and Development", ESPRIT BRWG 3264, Univ.Bremen, Bericht 6/89 (1989)
- [6] H.Ehrig, "Introduction to the Algebraic Theory of Graph Grammars", Lect. Notes in Comp. Sci. 73 (1979) 1-69
- [7] H.Ehrig, W.Fey, H.Hansen, M.Lowe, F.Parisi-Presicce, "Categories for the Development of Algebraic Module Specifications", Lect. Notes in Comp.Sci. 393 (1989) 157-184
- [8] H.Ehrig, W.Fey, F.Parisi-Presicce, E.K.Blum, "Algebraic Theory of Module Specifications with Constraint", invited, Proc MFCS, Lect. Notes in Comp. Sci. 233(1986) 59-77
- [9] H.Ehrig, H.-J.Kreowski, A.Maggiolo-Schettini, B.K.Rosen, J.Winkowski, "Transformation of Structures: An Algebraic Approach", *Math. Syst.Theory* 14 (1981) 305-334
- [10] H.Ehrig, B.Mahr, "Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics", EATCS Monographs on Theoret. Comp. Sci., vol 6, Springer-Verlag 1985
- [11] H.Ehrig, B.Mahr, "Fundamentals of Algebraic Specifications 2: Module Specifications and Constraints" EATCS Monographs on Theoret. Comp. Sci., to appear
- [12] H.Ehrig, F.Parisi-Presicce, "A Match for Rule Based Modular System Design", in preparation
- [13] H.Ehrig, H.Weber, "Algebraic Specification of Modules", in 'Formal Models in Programming' (E.J.Neuhold, G.Chronist, eds.) North-Holland 1985
- [14] H.Ehrig, H.Weber, "Programming in the large with Algebraic Module Specifications", in: H.J.Kugler(ed), Information Processing 1986, Amsterdam, North-Holland 1986/5, 675-684
- [15] J.A.Goguen, J.Meseguer, "Universal Realization, Persistent Interconnection and Implementation of Abstract Modules", Lect. Notes in Comp. Sci. 140 (1982) 265-281
- [16] F.Parisi-Presicce, "Union and Actualization of Module Specifications: Some Compatibility Results", *J. Comp. System Sci.*35,1 (1987) 72-95
- [17] F.Parisi-Presicce, "Partial Composition and Recursion of Module Specifications", Proc TAPSOFT87, Lect. Notes in Comp. Sci. 249 (1987) 217-231
- [18] F.Parisi-Presicce, "Product and Iteration of Module Specifications", Proc. CAAP 88, Lect. Notes in Comp. Sci. 299 (1988) 149-164
- [19] F.Parisi-Presicce, "Modular System Design applying Graph Grammar Techniques", Proc. 16 ICALP, Lect.Notes in Comp. Sci. 372 (1989) 621-636
- [20] D.Sannella, A.Tarlecki, "Toward Formal Development of Programs from Algebraic Specifications: Implementation revisited", Lect.Notes in Comp.Sci. 249 (1987) 96-110
- [21] D.Sannella, A.Tarlecki, "Toward Formal Development of ML Programs : Foundations and Methodology", U.Edinburgh Tech. Rep. ECS-LFCS-89-71
- [22] H.Weber, H.Ehrig, "Specification of Modular Systems", *IEEE Trans. on Soft. Eng. SE-12*, 7 (1986)784-798