DIAGRAMMES

M. C. GAUDEL
TH. MOINEAU

A theory of software reusability

Diagrammes, tome 23 (1990), p. 67-84
<http://www.numdam.org/item?id=DIA_1990__23__67_0>

© Université Paris 7, UER math., 1990, tous droits réservés.

L'accès aux archives de la revue « Diagrammes » implique l'accord avec les conditions générales d'utilisation (http://www.numdam.org/legal.php). Toute utilisation commerciale ou impression systématique est constitutive d'une infraction pénale. Toute copie ou impression de ce fichier doit contenir la présente mention de copyright.

NUMDAM
Article numérisé dans le cadre du programme
Numérisation de documents anciens mathématiques
http://www.numdam.org/

DIAGRAMMES VOLUME 23 , 1990

ACTES DES JOURNEES E.L.I.T.

(UNIV. PARIS 7, 27 JUIN-2 JUILLET 1988)

A THEORY OF SOFTWARE REUSABILITY *

M.C. GAUDEL [1] and Th. MOINEAU [1,2]

(1) Laboratoire de Recherche en Informatique
Unité associée au CNRS UA 410
Bât. 490, Université Paris-Sud
91405 ORSAY CEDEX
FRANCE

(2) SEMA GROUP
16 Rue Barbès
92126 MONTROUGE
FRANCE

Résumé :

La réutilisation du logiciel est un problème économique majeur et suscite de nombreux travaux. La plupart des approches actuelles de la réutilisation sont basées sur des méthodes empiriques, et aucune théorie formelle n'a encore été proposée. Ce papier donne une définition formelle de la réutilisabilité, basée sur des spécifications algébriques modulaires. Cette définition n'est pas totalement constructive, mais fournit une méthode pour trouver les composants potentiellement réutilisables et prouver leur réutilisation. Finalement les rapports entre réutilisation et hiérarchie sont examinés afin d'exploiter pleinement la réutilisation avec des spécifications hiérarchiques.

Mots-clés : réutilisation, types abstraits algébriques, spécifications formelles, modularité, PLUSS.

Introduction.

Software reusability is a topic of first practical importance. Most of the current approaches are based on empirical methods such as key words or descriptions in natural language. For some specific fields there exist good libraries of software components, and the description of the component is given in the terminology of the application area (mathematics, management, ...). However, there is no general approach to this problem.

To reuse a piece of software is only possible if **what** this piece of software does is precisely stated. It means that a specification of this software component is available. In this paper, we consider the case of software components which are formally specified using algebraic specifications. We deal with the following problem : given a specification SP' to be implemented, and a specification SP of an already implemented software component, is this

*This paper is a revised version of [GM 88].

reçu le 30/03/1990

component reusable for the implementation of SP' ? Thus we are not considering reusability of software design, or reusability of specification (which are also important problems) but reusability of code. A software component library should, at least, contain couples of the form <*formal specification, piece of code*>. Our claim is that the use of formal and structured specifications is fundamental for reusability. We define rigorously, in the case of algebraic structured specifications, the relations "is reusable for" and "is efficiently reusable for" between two specifications, the first one being already implemented. Moreover, it turns out that these definitions fit well with the primitives of our specification language.

We consider partial algebraic data types [GH 78,BW 82] and hierarchical specifications [WPPDB 83]. The specification language we use for our examples is PLUSS [Gau 85,Bid 89a] which is based on the ASL primitives [Wir 83]. The first part of the paper is a short presentation of these basic concepts. In part 2 we define, following [Bid 89b], the semantics of the use of predefined specifications. Part 3 is an informal introduction of our definitions of reusability. Part 4 states precisely these definitions. Part 5 explores the relationship between reusability and hierarchy : several theorems are given which show that our definitions are compatible both with the classical definitions of hierarchical specifications and with the practical aspects of software reusability.

Thus this paper suggests a criterion for software reusability which is theoretically founded. This criterion is not completely constructive, but it provides a guideline to find out reusable software components and prove their reuse. Moreover, the theorems of part 5 state how to exploit reusability in hierarchical specifications.

1 Basic definitions.

A **signature** $\Sigma = (S, F)$ consists of a set S of sort names and a set F of operation symbols, for each of which a profile $s_1 \dots s_n \longrightarrow s_{n+1}$ with $s_i \in S$ is given.

A **Σ-algebra** A is a family $(s^A)_{s \in S}$ of carrier sets together with a family of *partial* functions $(f^A)_{f \in F}$, such that the profiles of the operation names $f \in F$ coincide with the profiles of the functions f^A.

A (total) **Σ-morphism** ϕ from A to B is a family $(\phi_s)_{s \in S}$ of (total) applications $\phi_s : s^A \longrightarrow s^B$, such that for all operation name $(f : s_1 \dots s_n \longrightarrow s) \in F$ and all objects $a_1 \in s_1^A, \dots, a_n \in s_n^A$ the following holds : if $f^A(a_1, \dots, a_n)$ is defined then $\phi_s(f^A(a_1, \dots, a_n)) = f^B(\phi_{s_1}(a_1), \dots, \phi_{s_n}(a_n))$. Two Σ-algebra A and B are isomorphic, written $A \simeq B$, iff there is an isomorphism (bijective Σ-morphism) between them.

T_Σ is the well-known **term-algebra**. The interpretation t^A of a term t in a Σ-algebra A is specified by :

- if $t = c \in F$, then $t^A =_{def} c^A$.
- if $t = f t_1 \dots t_n$, then $t^A =_{def} f^A(t_1^A, \dots, t_n^A)$ provided that all the interpretations t_i^A and $f(t_1^A, \dots, t_n^A)$ are defined; otherwise t^A is undefined.

$T_{\Sigma \cup V}$ is the free Σ-algebra of terms with variables in V.

$PALG(\Sigma)$ is the category of the partial Σ-algebras with Σ-morphisms, $PGEN(\Sigma)$ is the category of finitely generated Σ-algebras (i.e. all elements of the carrier sets can be obtained by the interpretation t^A of a term $t \in T_\Sigma$).

A **specification** $SP = (\Sigma, E)$ consists of a signature Σ and a set E of *positive conditional axioms* on Σ : $\Phi_1 \wedge \dots \wedge \Phi_n \Longrightarrow \Phi_{n+1}$, where Φ_i are either equations $(t_1 = t_2)$ or definedness predicates $D(t)$ where t_1, t_2 and t are terms of $T_{\Sigma \cup V}$ (a Σ-algebra A satisfies $D(t)$ iff t^A is defined for all assignments of its variables).

$PALG(SP)$ is the category of all the models of SP (the Σ-algebras satisfying all axioms of E). $PGEN(SP)$ is those of finitely generated models. T_{SP} is the initial model of $PALG(SP)$ and $PGEN(SP)$. If $A \in PALG(SP)$, I_A is the unique Σ-morphism $I_A : T_{SP} \longrightarrow A$.

We use the abbreviation $SP_0 \subseteq SP$ for $\Sigma_0 \subseteq \Sigma$ ($S_0 \subseteq S$ and $F_0 \subseteq F$) and $E_0 \subseteq E$. We write $SP = SP_0 \cup \Delta SP$ for $\Sigma = \Sigma_0 \cup \Delta\Sigma$ ($S = S_0 \cup \Delta S$ and $F = F_0 \cup \Delta F$) and $E = E_0 \cup \Delta E$. ΔSP is called an **enrichment**. Note that these definitions are purely syntactic (union of presentations).

If $SP_0 \subseteq SP$, we note $U : PALG(\Sigma) \longrightarrow PALG(\Sigma_0)$ the **forgetful functor** defined by : $s^{U(A)} = s^A$ for $s \in S_0$ and $f^{U(A)} = f^A$ for $f \in F_0$. $U(A)$ corresponds to the Σ_0-reduct of A as defined in [WPPDB 83].

As mentioned in the introduction, we will deal with hierarchical specifications denoted by $SP = SP_0 + \Delta SP$ (this corresponds to the ENRICH construct of ASL).

2 Hierarchical models, implementations, realizations.

2.1 Hierarchical models.

As a preliminary to any study of reusability, it is necessary to define what is a correct implementation of a specification. This definition is dependent on the semantics considered for the specifications, i.e. the models associated with the specification. There are several approaches :

1. the models are those isomorphic to T_{SP}; such a semantics is called **initial semantics.**
2. the models are those isomorphic to the terminal algebra; such a semantics is called **terminal semantics.**
3. the models are all the models in $PALG(SP)$, such a semantics is called **"loose" semantics.**
4. a loose semantics may consider only finitely generated models, those in $PGEN(SP)$ [WPPDB 83].

Loose semantics makes it possible to give a simple definition of implementations [SW 82] : an algebra is an implementation of a specification SP, if, via some forgetting, restriction and identification, it is a model of SP.

However, loose semantics without hierarchy introduces trivial algebras among the models : the solution is to start from some basic specifications (such as booleans, naturals) with initial semantics, and to consider enrichments of specifications with hierarchical constraints.

Most of the time, it is convenient to consider finitely generated models : all the values are denotable by a term and thus are computable in a finite number of steps. Non finitely generated models are sometimes useful, but as we are concerned with reusability, it seems sound to choose finitely generated models : it means that the specification of a software component is supposed to mention all the functions of this components.

Thus we consider that the semantics of a specification SP is a subclass of $PGEN(SP)$ noted $HMOD(SP)$ of hierarchical finitely generated models, where the hierarchical constraints ensure that any model (implementation) of $SP = SP_0 + \Delta SP$ restricts into a model of SP_0. More precisely the forgetful functor from SP into SP_0 applied to a hierarchical model of SP gives a hierarchical model of SP_0.

Definition 2.1 *The class of the* **hierarchical models** *of a specification is :*

- $HMOD(SP_0) = \{T_{SP_0}\}$ *if* SP_0 *is a basic specification* (Σ_0, E_0).
- $HMOD(SP) = \{A \in PGEN(SP) | \ U(A) \in HMOD(SP_0)\}$ *if* SP *is a hierarchical specification :* $SP = SP_0 + \Delta SP$ *(*U *is the forgetful functor from* $PALG(\Sigma)$ *into* $PALG(\Sigma_0)$*).*

$HMOD(SP)$ is a full sub-category of $PGEN(SP)$. The existence of initial or terminal model is not ensured (T_{SP} is not always a hierarchical model [Ber 87]).

Of course, the class of models of a specification depends of its hierarchy; for instance :

$$HMOD((SP_0 + \Delta SP_1) + \Delta SP_2) \subseteq HMOD(SP_0 + (\Delta SP_1 \cup \Delta SP_2))$$

But the reverse is not always true. This will turn out to be quite important for reusability aspects.

We do not restrict the class of models to minimally defined ones [BW 82]. It means that an operation can be implemented in such a way that it is more defined than what is specified. This seems interesting in the framework of reusability.

2.2 Realization.

As stated in introduction, we consider couples <*specification, program*>, where the program is correct with respect to the specification. In the algebraic specification framework [EKMP 80,SW 82,BBC 86] it means that the program is a model of the specification modulo some forgettings, restrictions and identifications.

However, as soon as software is modular, we are concerned mainly with pieces of software which use other pieces of software. For instance, a module implementing a specification of the sets of integers includes some code for the operations insert and is-a-member, and uses some already existing implementation of the operations on integers. Thus, given a structured specification $SP = SP_0 + \Delta SP$ we call a "realization" of ΔSP a piece of software such that when coupling it with a hierarchical model (implementation) of SP_0, one gets a hierarchical model of SP. In order to be reusable, a realization of ΔSP must accept any model of SP_0 : no assumptions can be made on the implementation of SP_0. This property is not only important for reusability, but also in large software projects where several programmers are working concurrently on various modules.

These considerations lead to the following definition :

Definition 2.2 *A* **realization** *of enrichment* ΔSP *of* SP_0 *is a functor* Δ *from* $HMOD(SP_0)$ *into* $HMOD(SP_0 + \Delta SP)$ *such that :* $\forall A_0 \in HMOD(SP_0)$, $U(\Delta(A_0)) \simeq A_0$.

The class of realizations of ΔSP *on the top of* SP_0 *is noted* $REAL_{SP_0}(\Delta SP)$.

As stated above, this functor must be conservative. For instance, if SP_0 is an implementation of the integers by 8 bits strings, Δ will provide an implementation of SP where integers are implemented by 8 bits strings, not by 16 bits strings : for instance, there is no recoding of the integers by the realization of the sets of integers.

Composition of realizations works well :

Theorem 2.1 *Let* $SP_1 = SP_0 + \Delta SP_1$, $SP_2 = SP_1 + \Delta SP_2$ *and* $\Delta SP_3 = \Delta SP_1 \cup \Delta SP_2$.

If $\Delta_1 \in REAL_{SP_0}(\Delta SP_1)$ *and* $\Delta_2 \in REAL_{SP_1}(\Delta SP_2)$, *then* $\Delta_2 \circ \Delta_1 \in REAL_{SP_0}(\Delta SP_3)$.

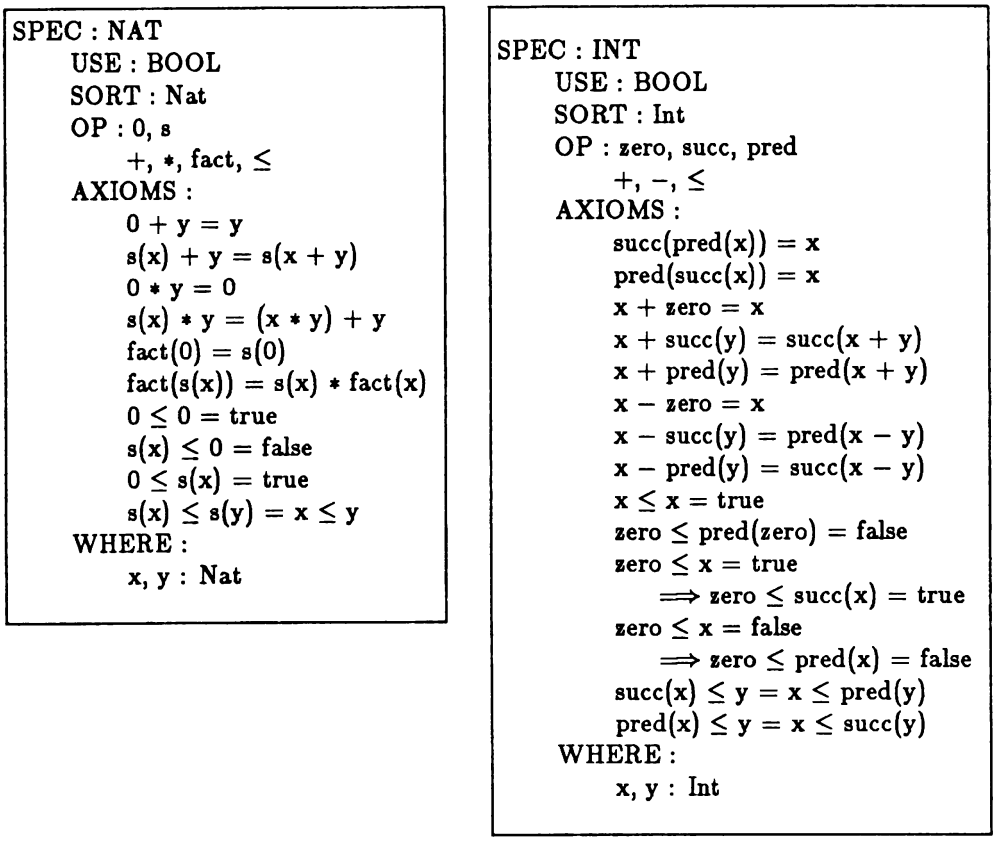


Figure 1: NAT and INT specifications

Proof : The forgetful functor U_3 from $ALG(SP_3)$ into $ALG(SP_0)$ is equal to $U_1 \circ U_2$ where U_1 et U_2 are the forgetful functors from $ALG(SP_1)$ into $ALG(SP_0)$ and from $ALG(SP_2)$ into $ALG(SP_1)$. Thus $\forall A_0 \in HMOD(SP_0) : U_3(\Delta_2 \circ \Delta_1(A_0)) = U_1 \circ (U_2 \circ \Delta_2) \circ \Delta_1(A_0) \simeq U_1 \circ \Delta_1(A_0) \simeq A_0$. □

It is important to note that there does not always exist a realization for a ΔSP. If ΔSP removes some models from $HMOD(SP_0)$, assuming for instance some implementation choices of SP_0, there exist no realization. However $SP = SP_0 + \Delta SP$ may have hierarchical models, thus global implementations. The existence of a realization means that the implementation choices in ΔSP and in SP_0 are completely independent.

3 What is reusability ?

3.1 Intuitive introduction.

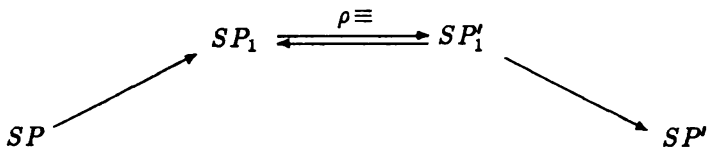


Figure 2: Scheme of reuse.

This part is an intuitive introduction to the formal definitions which are given in part 4.

Let us consider an example : suppose we have in a software components library a program corresponding to the specification INT of figure 1; we want to develop a program corresponding to the specification NAT of figure 1. Following our intuition and our experience of programming, it is obvious that the code of INT is reusable for implementing NAT. More precisely, BOOL, Nat, 0, s and $\leq$ can be implemented by reusing INT.

In this example, we notice that :

- a renaming is needed between INT and NAT;
- the axioms of INT and NAT are different;
- INT provides functions which are not required by NAT;
- some functions required by NAT are missing in INT.

These remarks lead to the scheme of figure 2 which shows that reusability of SP for SP' w.r.t. a subsignature Σ_r' of Σ' implies the existence of two specifications SP_1 and SP_1' such that :

1. SP_1 is an extension of SP : the missing functions (and possibly some hidden functions) are added to SP : $SP_1 = SP + \Delta SP_1$. Practically speaking, a realization of ΔSP_1 must be developed, since the reused program is only known by its specification SP.
2. SP_1' is an enrichment of SP' which is equivalent, modulo a renaming, to SP_1. This equivalence states the validity of the reuse of SP for SP' and makes it possible to have different axioms in SP and SP', as in example of figure 1. A more (and too) restrictive definition of reusability could consider only syntactic renaming from SP_1 to SP_1'.
3. Any model of SP_1' can be *restricted* via the forgetful functor to a model of SP' : unnecessary functions of SP, and hidden functions are forgotten :
$$\forall A_1' \in HMOD(SP_1'),\ U'(A_1') \in HMOD(SP')$$
4. In order to ensure that sorts and operations of Σ_r' are actually reused from SP, Σ_r' must be a renaming of a subsignature of Σ.

However condition 3 is too restrictive in some cases : if we consider the example of figure 1, implementing NAT by reuse of INT, the result of the forgetful functor is not a (hierarchical) finitely generated model of NAT, since the negative integers are kept. Thus we need a "stronger" forgetful functor than the classical one : by the way, it is a forget-restrict functor in the sense of [EKMP 80,SW 82]. Other problems arise if some functions are more defined in SP than in SP'. Part 4 gives the corresponding definitions.

3.2 Reusability vs abstract implementation.

Reuse is a special, simpler case of abstract implementation of algebraic data types [EKMP 80,SW 82,BBC 86]. In this paper we are concerned by direct reuse. It means that the "abstraction functions" of [BBC 86] or the "copy functions" of [EKMP 80] are just identities, and that the "identify function" is no more necessary to get a model of SP'.

4 Reusability : definitions.

As mentioned above, we have a reusability definition in three steps : extension, equivalence modulo a renaming, and forgeting-restricting. We study each of them successively.

4.1 Extension.

The extension phase consists in giving an enrichment ΔSP_1 on the top of SP, which must be realizable :

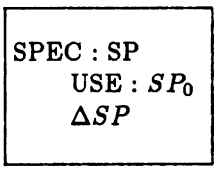
Definition 4.1 ΔSP *is* **realizable** *on the top of* SP_0 *iff* $REAL_{SP_0}(\Delta SP) \neq \emptyset$.

This property is noted $SP_0 \longrightarrow SP$ or $SP = SP_0 \oplus \Delta SP$.

Remark.

$REAL_{SP_0}(\Delta SP)$ is the semantics of the USE construct of the PLUSS specification language [Bid 89b]. PLUSS makes a distinction between the USE construct, which corresponds to the modular structure of the software, and the ENRICH construct which expresses the incremental development of specifications and which has the same semantics as in ASL.

$SP = SP_0 \oplus \Delta SP$ corresponds to the following construct in PLUSS :



Realizability is a transitive relation :

Theorem 4.1 *If* $SP_1 = SP_0 \oplus \Delta SP_1$ *and* $SP_2 = SP_1 \oplus \Delta SP_2$ *then* $SP_3 = SP_0 \oplus (\Delta SP_1 \cup \Delta SP_2)$.

Proof : Let $\Delta SP_3 = \Delta SP_1 \cup \Delta SP_2$. One can take $\Delta_1 \in REAL_{SP_0}(\Delta SP_1)$ and $\Delta_2 \in REAL_{SP_1}(\Delta SP_2)$. From theorem 2.1, $\Delta_2 \circ \Delta_1 \in REAL_{SP_0}(\Delta SP_3)$, and thus $REAL_{SP_0}(\Delta SP_3) \neq \emptyset$. □

Note that $HMOD(SP_3)$ is generally different from $HMOD(SP_2)$. SP_2 and SP_3 are equal if they are considered as non hierarchical specifications, but are different as hierarchical ones.

The definitions above does not cope with parameterization : Δ may perfectly take into account some properties of the specification SP_0. It must not take into account properties of the implementation of SP_0. Parameterized specifications and generic modules introduce additional difficulties and are not considered in this paper.

4.2 Equivalence of hierarchical specifications modulo a renaming.

A renaming between two signatures Σ_1 and Σ_2 is a signature isomorphism $\rho : \Sigma_1 \longrightarrow \Sigma_2$ [EM 85]. Some authors define renamings as injective signature morphisms [Pro 82]. Since we are interested in equivalence of specifications, we consider bijective renamings. Signature renamings easily extend into term renamings, algebra renamings, and (hierarchical) specification renamings.

The classical notion of specification equivalence is that SP and SP' are equivalent iff $\Sigma = \Sigma'$ and $T_{SP} \simeq T_{SP'}$. For hierarchical specifications, the definition is :

Definition 4.2 : hierarchical equivalence.

Two basic specifications SP_0 and SP'_0 are hierarchically equivalent iff $\Sigma_0 = \Sigma'_0$ and $T_{SP_0} \simeq T_{SP'_0}$.

Two hierarchical specifications $SP = SP_0 + \Delta SP$ and $SP' = SP'_0 + \Delta SP'$ are hierarchically equivalent iff :

- *SP_0 and SP'_0 are hierarchically equivalent*
- $\Delta\Sigma = \Delta\Sigma'$.
- $HMOD(SP) = HMOD(SP')$.

Hierarchical equivalence is denoted by $SP \equiv SP'$. It means that two hierarchical specifications are hierarchically equivalent iff they have the same (unfolded) signature, the same hierarchy, the same hierarchical models. Only the axioms can differ.

Proposition 4.2 *Let $SP = SP_0 + \Delta SP$ and $SP' = SP'_0 + \Delta SP'$. If $SP_0 \equiv SP'_0$, $\Delta\Sigma = \Delta\Sigma'$ and $T_{SP_0} \simeq T_{SP'_0}$, then $HMOD(SP_0 + \Delta SP) = HMOD(SP'_0 + \Delta SP')$ and thus $SP \equiv SP'$.*

Proof : by induction on the hierarchy. □

From the definitions above, it comes :

Definition 4.3 *Two hierarchical specifications SP and SP' are* **hierarchically equivalent modulo a renaming** *ρ ($SP \rho\equiv SP'$) iff $\rho(SP) \equiv SP'$.*

4.3 Forget-restrict functor.

As noted above, we need a forget-restrict functor in order to get finitely generated models of SP' (see fig. 2).

Definition 4.4 *Let $SP' \subseteq SP'_1$ two specifications.*

The **forget-restrict functor** *$V : PALG(SP'_1) \longrightarrow PGEN(SP')$ is defined by :*

- $\forall A \in PALG(SP'_1)$,
  - $\forall s \in S'$, $s^{V(A)} = [I_{U(A)}]_s(s^{T_{SP'}})$.
  - $\forall (f : s_1 \dots s_n \longrightarrow s) \in F'$,

$$f^{V(A)}(a_1, \dots a_n) = \begin{cases} f^{U(A)}(a_1, \dots, a_n) & \text{if } f^{U(A)}(a_1, \dots, a_n) \in s^{V(A)} \\ \text{undefined} & \text{otherwise} \end{cases}$$

- *$\forall \phi : A \longrightarrow B$, $V(\phi)$ is the restriction of $U(\phi)$ to $V(A)$.*

As defined above, $I_{U(A)}$ is the unique morphism from $T_{SP'}$ into $U(A)$.

V(A) is the finitely generated part of $U(A)$ with respect to Σ' ($s^{V(A)} = \{a \in s^A \mid \exists t \in T_{SP'},\ a = t^A\}$). By the way, the sorts of SP' are subsorts of those of SP'_1 [FGJM 85]. It's easy to prove that V is a functor from $PALG(SP'_1)$ into $PGEN(SP')$, and that composition of forget-restrict functors works well : if V_2 is the forget-restrict functor from SP_2 to SP_1 and V_1 is the forget-restrict functor from SP_1 to SP_0 then $V_1 \circ V_2$ is the forget-restrict functor from SP_2 to SP_0.

Besides, by definition, the forget-restrict functor restricts the domain of functions in such a way that only finitely generated objects can be got as result.

4.4 Reusability.

We now put together the three steps and define reusability.

Definition 4.5 *SP is* **reusable for SP' w.r.t. a signature Σ'_r modulo a renaming** *ρ ($SP\ \rho\hookrightarrow SP'[\Sigma'_r]$) iff there exist two specifications SP_1 and SP'_1 such that :*

[R1] $SP_1 = SP \oplus \Delta SP_1$.

[R2] $SP_1\ \rho\equiv SP'_1$.

[R3] $SP' \subseteq SP'_1$.

[R4a] $\forall A'_1 \in HMOD(SP'_1),\ V'(A'_1) \in HMOD(SP')$.

[R5] $\Sigma'_r \subseteq \Sigma'$ *and* $\rho^{-1}(\Sigma'_r) \subseteq \Sigma$.

where V' is the forget-restrict functor from $PALG(SP'_1)$ into $PGEN(SP')$.

Note that in A'_1, some functions can be more defined than what is required by SP'. It does not matter since $HMOD(SP')$ is not limited to minimally defined models (as defined in [BW 82]).

Definition 4.6 *SP is* **efficiently reusable for SP' w.r.t. a signature Σ'_r modulo a renaming** *ρ ($SP\ \rho\rightsquigarrow SP'[\Sigma'_r]$) iff there exist two specifications SP_1 and SP'_1 such that :*

[R1] $SP_1 = SP \oplus \Delta SP_1$.

[R2] $SP_1\ \rho\equiv SP'_1$.

[R3] $SP' \subseteq SP'_1$.

[R4b] $\forall A'_1 \in HMOD(SP'_1),\ U(A'_1) \in HMOD(SP')$.

[R5] $\Sigma'_r \subseteq \Sigma'$ *and* $\rho^{-1}(\Sigma'_r) \subseteq \Sigma$.

where U' is the forgetful functor from $PALG(SP'_1)$ into $PALG(SP')$.

The intuitive notion behind efficient reusability is that the carriers of SP are finitely generated with respect to SP' : there are no useless values.

At the practical level, our definition of reusability in three steps will result in specifications of the form :

SPEC : X **FORGET** ...
(**USE** : SP
ΔSP_1) **RENAMING** ... **INTO** ...
END X

Note that X is an abstract implementation of SP' : $HMOD(X) \subseteq HMOD(SP')$.

The second step (equivalence modulo a renaming) can be skipped : SP_1 and SP_1' can be embedded in an unique but more complex specification SP_2 :

Theorem 4.3 *SP is reusable (resp. efficiently reusable) for SP' w.r.t. Σ_r' modulo a renaming ρ iff there exists a specification SP_2 such that :*

[1] $SP_2 = SP \oplus \Delta SP_2$.

[2] $SP' \subseteq \rho(SP_2)$.

[3] $\forall A_2 \in HMOD(SP_2)$, $V'(\rho(A_2)) \in HMOD(SP')$ *(resp.* $U'(\rho(A_2)) \in HMOD(SP')$*).*

[4] $\Sigma_r' \subseteq \Sigma'$ *and* $\rho^{-1}(\Sigma_r') \subseteq \Sigma$.

Proof : $\Leftarrow$ Take $SP_1 = SP_2$ and $SP_1' = \rho(SP_2)$.
$\Rightarrow$ Take $SP_2 = (\Sigma_1, E_1 \cup \rho^{-1}(E_1'))$. Since $SP_1\ \rho \equiv SP_1'$ the axioms of SP_1' (renamed by ρ^{-1}) do not change the hierarchical models of SP_1. □

This approach is less close to reality, but easier to deal with theoretically. Consequently, in the rest of this paper we will consider reuses in two steps. Besides, we ignore renaming, since it is only a matter of syntactic sugar. However, the results still hold in the case of reuse with a renaming. Reusability and efficient reusability without renaming are noted $\hookrightarrow$ and $\rightsquigarrow$. We will denote a specific reuse of SP for SP' by the scheme : $SP \longrightarrow SP_1 \searrow^V SP'[\Sigma_r']$, and an efficient reuse by $SP \longrightarrow SP_1 \searrow^U SP'[\Sigma_r']$.

Properties.
- if SP is efficiently reusable for SP' w.r.t. Σ_r' then SP is reusable for SP' w.r.t. Σ_r'.
- if $SP \longrightarrow SP'$ then SP is efficiently reusable for SP' w.r.t. Σ.

Example.

The following example illustrates these definitions. We consider four specifications and look at the relations of reusability between them.

SPEC : NAT1
USE : BOOL
SORT : Int
OP : 0, s, + , *
AX : ...

SPEC : NAT2
USE : BOOL
SORT : Int
OP : 0, s, $\leq$
AX : ...

SPEC : INT1
USE : BOOL
SORT : Int
OP : 0, s, p
AX : ...

SPEC : INT2
USE : BOOL
SORT : Int
OP : 0, s, p, $\leq$
AX : ...

The axioms are the classical ones and are omitted. The hierarchical models are :
- $HMOD(\text{NAT1}) = \{N\} \cup \{Z/nZ \mid n \in N\}$
- $HMOD(\text{NAT2}) = \{N\}$
- $HMOD(\text{INT1}) = \{Z\} \cup \{Z/nZ \mid n \in N\}$
- $HMOD(\text{INT2}) = \{Z\}$

Thus :
- NAT2 is efficiently reusable for NAT1 w.r.t. $\Sigma_{BOOL} \cup \langle\{Int\}, \{0, s\}\rangle$.
- INT1 is reusable for NAT1 w.r.t. $\Sigma_{BOOL} \cup \langle\{Int\}, \{0, s\}\rangle$.
- INT2 is reusable for NAT2 w.r.t. $\Sigma_{BOOL} \cup \langle\{Int\}, \{0, s, \leq\}\rangle$.

But :
- NAT1 is not reusable for NAT2 w.r.t. $\Sigma_{BOOL} \cup \langle\{Int\}, \{0, s\}\rangle$. Since $Z/2Z$ is a NAT1 model for which there is no extension into N, and N is the only model of NAT2.
- NAT1 is not reusable for INT1 w.r.t. $\Sigma_{BOOL} \cup \langle\{Int\}, \{0, s\}\rangle$.
- NAT2 is not reusable for INT2 w.r.t. $\Sigma_{BOOL} \cup \langle\{Int\}, \{0, s, \leq\}\rangle$.

□

The reusability and efficient reusability relations are not symmetric (see NAT1 and INT1 above). This is not surprising as soon as we want integers to be reusable for the naturals but not the reverse. The transitivity of these relations is discussed below.

5 Reusability, reuse and hierarchy.

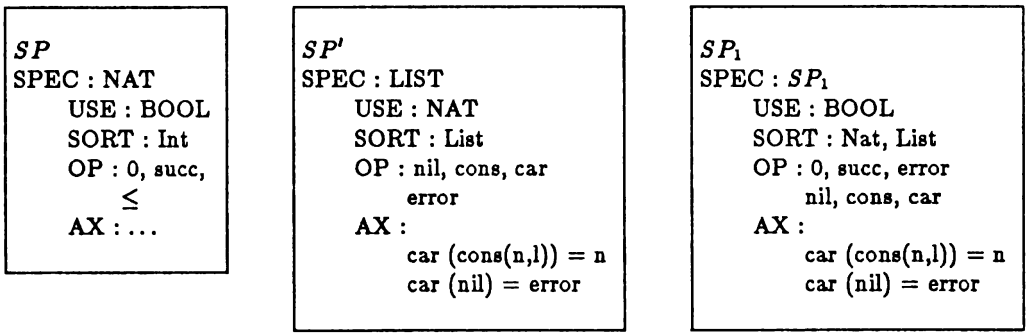
Our formal definition of reusability is applicable only if it is compatible with the primitives of most of the specification languages. This part of the paper discusses the relationship between reusability, reuse and hierarchy.

5.1 What about primitive specifications ?

The first result we give is negative :

Fact 5.1 *There exist specifications SP and SP' such that $SP = SP_0 \oplus \Delta SP$ and $SP \hookrightarrow SP'[\Sigma_r]$ (resp. $SP \rightsquigarrow SP'[\Sigma_r]$) but $SP_0 \not\hookrightarrow SP'[\Sigma_r \cap \Sigma_0]$ (resp. $SP_0 \not\rightsquigarrow SP'[\Sigma_r \cap \Sigma_0]$).*

Counter-example : Consider the specifications (here SP_0 is the specification BOOL) :



SP is reusable for SP' w.r.t. Σ_{NAT}, but SP_0 is not reusable for SP' w.r.t. Σ_{BOOL} via SP_1. For the Σ_1-algebra A_1 such that $Int^{A_1} = N \cup \{error\}$ and $(error \leq n) = true$ is a hierarchical model of SP_1. But A_1 is not a hierarchical model of SP' since $U'(A_1)$ is not finitely generated with respect to Σ_{NAT} (cf. *error*).

□

This result seems surprising : if SP is reusable for SP', it seems tempting to reuse a primitive part of SP for SP'. It is well known that as soon as we consider hierarchical specifications, hierarchy is of first importance. This result exemplifies this importance : one may not modify the hierarchy, or ignore it, without care.

It is clear that, if it is possible to flatten the specifications without modification of the hierarchical models, there is no problem. However, fact 5.1 points out that reusability is not generally transitive. Fortunately we will see in part 5.2 that reuse is nevertheless compatible with hierarchy.

Our second result is a kind of symmetric one :

Theorem 5.2 *If* $SP \hookrightarrow SP'[\Sigma_r]$ *(resp.* $SP \rightsquigarrow SP'[\Sigma_r]$*) and* $SP' = SP'_0 \oplus \Delta SP'$*, then* $SP \hookrightarrow SP'_0[\Sigma_r \cap \Sigma'_0]$ *(resp.* $SP \rightsquigarrow SP'_0[\Sigma_r \cap \Sigma'_0]$*).*

Proof : straightforward □

5.2 What about enrichment (efficient case).

Let us consider now the case when SP is efficiently reusable for SP' via SP_1. What can be done for implementing $SP'' = SP' \oplus \Delta SP''$?

Theorem 5.3 *Reusability of enrichment.*

$$\begin{array}{ccc} HMOD(SP_2) & \xrightarrow{U''_2} & HMOD(SP'') \\ \uparrow \overline{\Delta} & & \uparrow \Delta \\ HMOD(SP_1) & \xrightarrow{U'_1} & HMOD(SP') \end{array}$$

Let SP, SP_1 *and* SP' *be specifications such that* $SP \longrightarrow SP_1 \searrow^U SP'$ *(*SP *is efficiently reusable for* SP' *w.r.t. some* Σ'_r*), and* SP'' *a specification which use* SP' *:* $SP'' = SP' \oplus \Delta SP''$.

If $\Sigma_1 \cap \Delta\Sigma'' = \emptyset$*, then* $SP_2 = SP_1 \oplus \Delta SP''$ *and for all realization* Δ *of* $\Delta SP''$ *on top of* SP'*, there exists a realization* $\overline{\Delta}$ *of* $\Delta SP''$ *on top of* SP_1*, such that the above diagram commute, i.e. :*

$$\forall A_1 \in HMOD(SP_1),\ U''_2 \circ \overline{\Delta}(A_1) \simeq \Delta \circ U'_1(A_1)$$

Proof : See annex A □

This theorem is a generalization of the extension lemma given in [EM 85] for equational specifications and total, initial algebras.

Figure 3 shows the difference between the situations studied in fact 5.1 and theorem 5.3. By the way theorem 5.3 deals with another kind of reusability : those of the realizations of $\Delta SP''$. Moreover it shows that $\Delta SP''$ can be realized independently of the reuse done for SP'.

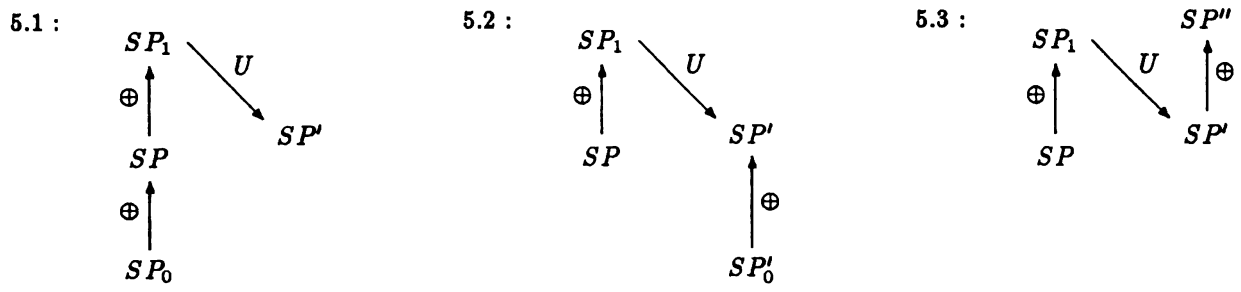


Figure 3: Situations of fact 5.1 and theorems 5.2 and 5.3

The proof of the theorem 5.3 is constructive : it shows how to get a realization of $\Delta SP''$ on the top of SP_1 given a realization of $\Delta SP''$ on the top of SP' : just putting together the carriers and the operations of A_1 (which is an implementation of SP_1) and of A''. Practically it corresponds to putting together type declarations and code of operations.

This result is important : it allows to consider a program as a set of pieces, the bodies of which can be replaced or developed in an independent way. For instance, let us suppose that we have to develop the components below :

SPEC : L_NAT USE : NAT SORT : List OP : nil, cons, ... AX : ...	SPEC : NAT USE : BOOL SORT : Nat OP : 0, succ, +, *, $\leq$ AX : ...

and we already have programs implementing :

SPEC : L_NAT1 USE : NAT1 SORT : List OP : nil, cons, ... AX : ...	SPEC : NAT1 USE : BOOL SORT : Nat OP : 0, succ, $\leq$ AX : ...	SPEC : NAT2 USE : BOOL SORT : Nat OP : 0, succ, +, *, $\leq$ AX : ...

We know, from theorem 5.3, that it is possible to replace the NAT1 component by the NAT2 component in L_NAT1, and to use L_NAT1 (with NAT2) to implement L_NAT. This avoids to implement addition and multiplication on naturals.

Remark. The condition $\Sigma_1 \cap \Delta\Sigma'' = \emptyset$ is essential; it is not a problem to satisfy it using appropriate renaming of what is forgotten from SP_1 to SP'. □

SPEC : SET
USE : NAT
SORT : Set
OP : empty, insert, is-in
AX :
(1) is-in(x,empty) = false
(2) x = y ⇒ is-in(x,insert(y,s)) = true
(3) x ≠ y ⇒ is-in(x,insert(y,s)) = is-in(x,s)
(4) is-in(x,s) = true ⇒ insert(x,s) = s
(5) (is-in(x,s) = false) ∧ (x ≤ 0 = true)
⇒ insert(x,s) = insert(0,s)

```
(defun empty ()
       '())
(defun is-in (n set)
       (member n set))
(defun insert (n set)
       (if (is-in n set)
           set
           (if (<= n 0)
               (insert 0 set)
               (cons n set))))
```

Figure 4: A specification and a (strange) realization of sets of naturals numbers.

5.3 What about enrichment (non efficient case).

It would be nice to extend the previous results to the non efficient reusability, unfortunately the previous theorem is no more valid. For instance, let us consider the specifications NAT and INT of figure 1 and the specification SET of the figure 4 (the last axiom looks strange, but it is on purpose).

INT is not efficiently reusable for NAT; SET = NAT⊕ΔSET, but INT+ΔSET has no hierarchical models since :

true	=	is-in(-1,insert(-1,empty))	from (2)
	=	is-in(-1,insert(0,empty))	from (5)
	=	is-in(-1,empty))	from (3)
	=	false	from (1)

By the way, the problem is more fundamental than it seems : suppose now that the strange axiom (5) is removed. The LISP program of figure 4 is a correct (but strange) realization of sets of natural numbers. However, it is no more the case if naturals are replaced by integers. Then, the only way to get such a realization is to define a predicate `is-a-nat` and to add a check of this predicate in front of each function. Unfortunately, the introduction of such predicates (i.e. subsorts) makes the reuse process much more complicated.

Definition 5.1 *Let $SP_0 \subseteq SP$ two specifications which use a specification BOOL of booleans. A* **discriminant predicate** *between SP and SP_0, for $s \in S_0$, is an operation $(p_s : s \longrightarrow Bool) \in F$ such that :*

$$\forall A \in PGEN(SP),\ \forall a \in s^A,\ p_s^A(a) = \begin{cases} true & if\ a \in s^{V(A)} \\ false & otherwise \end{cases}$$

If $s \in S \backslash S_0$, by convention, $p_s^A(a) = false, \forall a \in s^A$.

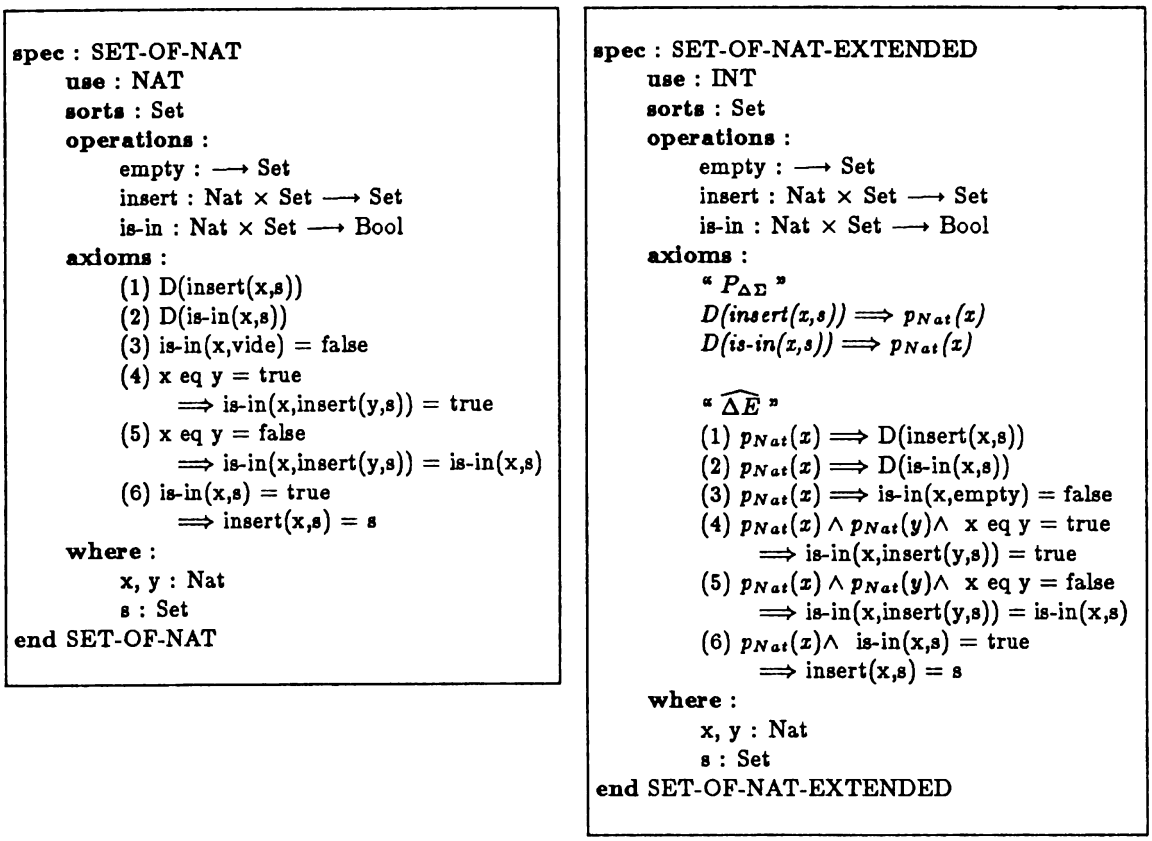


Figure 5: A example of extended enrichment.

Remark.

A discriminant predicate is the characteristic function of the finitely generated part of ${}_s{}^{V(A)}$. If SP and SP_0 have free generators C and C_0 ($C_0 \subseteq C$), the definition of the p_s predicates is straightforward :

- $p_s(c_0) = true$ for all constant $(c_0 :\longrightarrow s) \in C_0$.
- $p_{s_1}(x_1) = true \wedge \ldots \wedge p_{s_n}(x_n) = true \iff p_s(f_0(x_1 \ldots x_n)) = true$ for all $(f_0 : s_1 \ldots s_n \longrightarrow s) \in C_0$.
- $p_s(c) = false$ for all constant $(c :\longrightarrow s) \in C \backslash C_0$.
- $p_s(f(x_1 \ldots x_n)) = false$ for all $(f : s_1 \ldots s_n \longrightarrow s) \in C \backslash C_0$.

If it is not the case, the definition of these predicates is highly dependent on the rest of the specification.

□

Definition 5.2 : extended enrichment.

Let $PR_0 \subseteq PR_1$ be two presentations, ΔSP an enrichment of PR_0, such that $\Sigma_1 \cap \Delta\Sigma = \emptyset$, and $(p_s)_{s \in S_1}$ the discrimant predicates between PR_0 and PR_1.

An extended enrichment of ΔSP from PR_0 to PR_1, is the enrichment $\overline{\Delta SP} = (\Delta\Sigma, \widehat{\Delta E} \cup P_{\Delta\Sigma})$ where :

- $P_{\Delta\Sigma}$ *is the set of axioms :* $D(f(x_1,\ldots,x_i,\ldots,x_n)) \Longrightarrow p_{s_i}(x_i) = true$, *for each* $(f : s_1 \ldots s_n \longrightarrow s) \in \Delta F$ *and* $s_i \in S_0$.
- $\widehat{\Delta E}$ *is obtained from* ΔE *by the following transformations : any axiom* $\Phi_1 \wedge \ldots \wedge \Phi_n \Longrightarrow \Phi$ *becomes :* $\bigwedge_i [p_{s_i}(x_i) = true] \wedge \Phi_1 \wedge \ldots \wedge \Phi_n \Longrightarrow \Phi$, *where the* x_i *are all the variables of sort* $s_i \in S_0$ *that occurs in the* Φ_j *and* Φ.

The $P_{\Delta\Sigma}$ axioms express that the operations of ΔSP, when SP_1 is used instead of SP_0, must be restricted to the parts of the sorts of SP_1 which are finitely generated by Σ_0. Similarly, the Φ premisses added to the axioms ΔE express that these axioms are valid on these parts of the sorts.

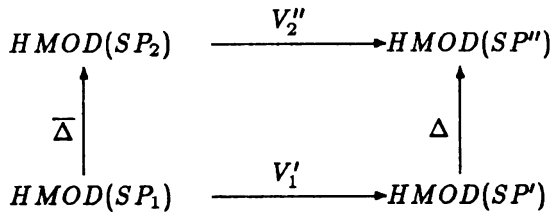
Example.

The figure 5 shows how an enrichment SET-OF-NAT which uses NAT (cf. Figure 1) can be transformed into SET-OF-NAT-EXTENDED which uses INT, using the predicates `greater-or-equal-to-zero` as p_{Nat} and with p_{Bool} defined by : $p_{Bool}(x) = true$ iff $x = true$ or $x = false$.

It is important to note that we reuse INT for NAT : thus we get a specification of sets of natural numbers which uses INT. It is not a specification of sets of integers.

We have a theorem similar to theorem 5.3 :

Theorem 5.4 *Reusability of enrichment.*



Let SP, SP_1 *et* SP' *be specifications such that* $SP \longrightarrow SP_1 \searrow^V SP'$ *(*SP *is reusable for* SP' *w.r.t. some* Σ'_r*), and* $SP'' = SP' \oplus \Delta SP''$, *with* $\Sigma_1 \cap \Delta\Sigma'' = \emptyset$.

Then $SP_2 = SP_1 \oplus \overline{\Delta SP''}$, *and for all realization* $\Delta \in REAL_{SP'}(\Delta SP'')$, *there is a realization* $\overline{\Delta} \in REAL_{SP_1}(\overline{\Delta SP''})$, *such that the diagram above commutes, i.e. :*

$$\forall A_1 \in HMOD(SP_1),\ V_2'' \circ \overline{\Delta}(A_1) \simeq \Delta \circ V_1'(A_1)$$

Proof : See annex B □

Conclusions and further researches.

We have given a criterion for "efficient" software reusability, and proved that this notion of reusability fits well for hierarchical specifications. We have introduced "non efficient reusability" (such as integers for naturals) in order to be more permissive. However, the results on this kind of reusability are slightly disappointing since they seem rather complex to apply.

It is interesting to note that non efficient reusability is not exactly what we have called "direct reusability", and is a step forward implementation. These results enforce our opinion that the kind of reusability we are studying, i.e. code reusability using its specification, should be done on as-it-is bases whenever possible. This is quite coherent with practice.

This study was devoted to structured specification and modular programs. It is clear that it must be extended to parameterized specifications and generic programs.

Acknowledgements.

Our thanks to Michel Bidoit and Gilles Bernot for numerous fruitful discussions and friendly encouragements.

This work is partially funded by ESPRIT (Meteor Project) and by the PRC-Greco "Programmation et outils pour l'intelligence artificielle". Th. Moineau's grant is funded by Sema-Metra.

References

[BBC 86] G. Bernot, M. Bidoit and C. Choppy, "Abstract implementation and correctness proofs", in *Proc. 3rd STACS*, Jan. 1986, Springer-Verlag LNCS 210, Jan. 1986.

[Ber 87] G. Bernot, "Good functors ... are those preserving philosophy !", in *Proc. 2nd Summer Conference on Category Theory and Computer Science*, Edinburgh, Sept. 1987.
also *LRI report* No. 354, June 1987.

[Bid 89a] M. Bidoit, "Pluss, un langage pour le développement de spécifications algébriques modulaires", Thèse d'Etat, Université Paris-Sud, Mai 1989.

[Bid 89b] M. Bidoit, "The stratified loose approach : A generalization of initial and loose semantics", in *Proc. 1st Int. Conf. on Algebraic Methodology and Software Technology*, Iowa City, USA, May 1989.

[BW 82] M. Broy and M. Wirsing, "Partial abstract types", *Acta Informatica*, No. 18, 1982.

[EKMP 80] H. Ehrig, H. Kreowski, B. Mahr and P. Padawitz, "Algebraic implementation of abstract data types", *Theoretical Computer Science*, Oct. 1980.

[EM 85] H. Ehrig and B. Mahr, "Fundamentals of algebraic specification", Springer Verlag, Berlin-Heidelberg-New York-Tokyo, 1985.

[FGJM 85] K. Futatsugi, J.A. Goguen, J-P. Jouannaud and J. Meseguer, "Principles of OBJ2", in *proc. 12th ACM Symposium on Principles of Programming Languages*, Jan. 1985.

[Gau 85] M.-C. Gaudel, "Towards structured algebraic specifications", *ESPRIT'85 - Status Report*, Part I, pp. 493–510, North Holland, 1986.

[GM 88] M.-C. Gaudel and Th. Moineau, "A theory of software reusability", In *Proc. ESOP'88*, LNCS 300, Springer Verlag, 1988.

[GH 78] J.V. Guttag and J.J. Horning, "The algebraic specification of abstract data types." *Acta Informatica*, No. 10, 1978.

[Pro 82] K. Proch, "ORSEC : Un Outil de Recherche de Spécifications Equivalentes par Comparaison d'exemple", Thèse de 3eme cycle, Nancy I, Dec. 1982.

[SW 82] D. Sanella and M. Wirsing, "Implementation of parametrized specifications", *Report CSR-102-82*, Department of Computer Science, University of Edinburgh.

[Wir 83] M. Wirsing, "Structured algebraic specifications : a kernel language", Habilitation Thesis, Technische Universität München, Sept. 1983.

[WPPDB 83] M. Wirsing, P. Pepper, H. Partsch, W. Dosch and M. Broy, "On hierarchy of abstract data types", *Acta Informatica*, No. 20, 1983.