A module concept 
within the initial behaviour framework

M.P. Nivela
F. Orcjas

Report LSI–88–23
Abstract: A module concept is defined which enables the description of the whole design of software systems, that is, not only the specification of problem requirements but also the specification of the process of implementation design. Moreover, this concept encapsulates (like in packages and similar constructions) both the interface and the implementation parts.

This module concept is based on a behaviour approach where the notion of implementation coincides with realization of the behaviour, that is, an abstract data type implements another if reproduces its behaviour. For that, a special kind of signature morphism, called implementation morphism, is introduced in order to formalize this realization of a behaviour idea. It is shown that implementation defined in this way is compatible with parameter passing.

Resum: En aquest paper s'introdueix un concepte de mòdul que permeteix la descripció del disseny complet de sistemes de software, és a dir, no solament la especificació dels requeriments del problema sinó també la especificació del procés de disseny de la implementació. Aquest concepte encapsula (com en els packages i en altres construccions similars) la part de interface i la part de implementació.

El concepte de mòdul que es presenta està basat en semàntica de comportament, facilitant una formalització més natural del concepte de implementació (que coincideix amb el concepte de realització de un comportament) mitjançant un tipus especial de morfisme de signatures anomenat morfisme de implementació.

D'aquesta manera es pot demostrar la compatibilitat entre la implementació i el pas de paràmètres.
A MODULE CONCEPT WITHIN THE
INITIAL BEHAVIOUR FRAMEWORK

Mª Pilar Nivela
Fernando Orejas
Facultad de Informática
Barcelona
Spain

(Preliminary version)

Abstract

A module concept is defined which enables the description of the whole design of software systems, that is, not only the specifications of problem requirements but also the specification of the process of implementation design. Moreover, this concept encapsulates (like in packages and similar constructions) both the interface and implementation parts.

This module concept is based on a behaviour approach where the notion of implementation coincides with realization of the behaviour, that is, an abstract data type implements another if reproduces its behaviour. For that a special kind of signature morphism, called implementation morphism, is introduced in order to formalize this realization of a behaviour idea and it is shown that implementation defined in this way is compatible with parameter passing.

1. Introduction

The basic ideas underlying software design methodology are

- to specify the behaviour of every abstraction identified in the problem, that is, to explain what the abstraction does.
- to hide the details of a specific implementation, that is, how the abstraction is realized.

This implies that, when selecting an implementation, one can choose any other abstraction with a behaviour such that it is equivalent to the one specified. Moreover, some hiding process is needed in order to be able to handle hiding information.

So, the algebraic approach to software modules has to take into account a behavioural equivalence concept and a change of observability mechanism.
The model concept presented in this paper corresponds to the modular constructions which appear in programming languages: packages, clusters, etc. A module M consists of four parts. One of them is a parameterized specification which determines the interface of the module, that is, all what the module offers to a feasible user. We will designed by interface(M). The second and third parts constitute the implementation of the interface. Exactly, the second one, designed by impl(M), is what it is "imported" (another module) to be able to carry out the implementation. The implementation part is again a module, that is, it has an interface specification and an implementation part. The third part of a module M, designed by enrich(M), is the enrichment of the imported module needed in the implementation. Finally, a module contains the implementation morphism, impl_morph(M), which establishes the connection between the interface and its implementation. This connection allows to associate an observable sort to a non-observable sort. In this manner the implementation of non-observable objects will be made in terms of observable sorts. The sorts of the predefined modules in the specification environment have all their sorts as observable.

Moreover, this module notion integrates enrichment, parameterization and parameter passing notions to be able to really design software systems.

A kind of such modules can be added to a specification language to transform it into a specification/design language, because in this way it would be possible to specify all the design of a problem.

The paper is organized as follows. The second section is an overview of the behaviour framework on which is based the rest of the concepts introduced later. In the third section the definition of implementation is defined using the concept of implementation morphism. Section 4 is related with the definition of behaviour module and some constructions like enrichment, union and actualization are defined.

2. Semantics of behaviour specifications

This section is a resume of the initial behaviour framework and the complete development can be found in [Ni 87, NiOr 87].

2.1 Behaviour specifications and semantics of a non parameterized behaviour specification

The criterion to behaviourally compare models is established as usual by means of a set of sorts called observable sorts. A behaviour signature Sig is a triple Sig = (Obs, S, Σ) with Obs ⊆ S. The sorts of Obs are called observable sorts.

A computation over an algebra A is a term of $T_Σ(A_{Obs})$ and an observable computation over A is a term of $T_Σ(A_{Obs})_s$ with $s \in$ Obs. These computations over A have an evaluation over A, called $ε_Λ$, which is the unique $Σ$-homomorphism extending the inclusion of $A_{Obs}$ into A.

The category of models we consider is Beh(Sig) defined in the following way:
Definition 2.1.1
The category \( \text{Beh}(\text{Sig}) \) has as objects all the \( \Sigma \)-algebras and a morphism \( f : A \to B \) is a family of functions \( f_s = \{ f_s^s \}_{s \in \text{Obs}} \) such that for all observable computations \( t \in T\Sigma(\text{Obs})_s \), \( s \in \text{Obs} \), it holds \( f_s (e_A, s(t)) = e_B, f^s(t) \), where \( f^# : T\Sigma(\text{Obs})_s \to B \) is the unique \( \Sigma(A_{\text{Obs}})_s \)-homomorphism which extends \( f \).

That means that the diagram

\[
\begin{array}{ccc}
T \Sigma(A_{\text{Obs}}) & \xrightarrow{f^#} & T \Sigma(B_{\text{Obs}}) \\
\varepsilon_A \downarrow & & \downarrow \varepsilon_B \\
A & \xrightarrow{f} & B
\end{array}
\]

commutes for every observable sort.

A morphism in this category is called a \( \Sigma \)-behaviour morphism. We will write \( \Sigma \)-behaviour morphism if \( \text{Sig} \) is clear from the context. A \( \Sigma \)-behaviour morphism \( f : A \to B \) between two \( \Sigma \)-algebras \( A \) and \( B \) is a \( \Sigma \)-behaviour isomorphism if \( f_s \) is bijective for every \( s \in \text{Obs} \).

It is said that \( A \) and \( B \) are behaviourally equivalent if there exists a \( \Sigma \)-behaviour morphism \( f : A \to B \) which is a \( \Sigma \)-behaviour isomorphism. We will denote that by \( A \equiv_{\Sigma} B \) (or \( A \equiv B \) when any possible confusion can arise about the signature \( \text{Sig} \)).

The satisfaction of a \( \Sigma \)-equation \( e : t_1 = t_2 \) of sort \( s \) by a \( \Sigma \)-algebra \( A \) is behavioural satisfaction, that is, \( A \) behaviourally satisfies \( e \) if \( A \) satisfies \( c(\alpha(e)) \) for every \( \Sigma \)-context \( c[z] \) over the sort \( s \) and every assignment \( \sigma : \text{var}(e) \to T\Sigma(X_{\text{Obs}}) \), where a \( \Sigma \)-context over the sort \( s \) is a term \( c[z] \) of \( T\Sigma(X_{\text{Obs}} \cup \{z\})_s \) with \( s' \in \text{Obs} \) and \( z \) a variable symbol of sort \( s \) not belonging to \( X \).

We will indicate that by \( A \models_B e \). That is equivalent to say that \( A \) behaviourally satisfies \( e \) if \( e_A(c_A(\alpha(t_1))) = e_A(c_A(\alpha(t_2))) \), for every \( \Sigma \)-context \( c_A[z] \) for \( A \) over the sort \( s \) and for every assignment \( \alpha : \text{var}(e) \to T\Sigma(A_{\text{Obs}}) \), where a \( \Sigma \)-context for a \( \Sigma \)-algebra \( A \) over the sort \( s \) is a term \( c_A[z] \) of \( T\Sigma(A_{\text{Obs}} \cup \{z\})_{s'} \) with \( s' \in \text{Obs} \) and \( z \) a variable symbol of sort \( s \) not belonging to \( X \).

A behaviour specification \( \text{SP} \) is a 4-tuple, \( \text{SP} = (\text{Obs}, S, \Sigma, E) \), where \( \text{Sig} = (\text{Obs}, S, \Sigma) \) is a behaviour signature and \( E \) is a set of \( \Sigma \)-equations and \( \text{Beh}(\text{SP}) \) is the full subcategory of \( \text{Beh}(\text{Sig}) \) of all \( \Sigma \)-algebras which behaviourally satisfy the equations of \( E \).

The semantics of a behaviour specification \( \text{SP} \) is the initial behaviour, that is, the behaviour of \( T_{\text{SP}} \). It holds that this behaviour is initial among all the behaviour, that is, if \( A \) is a \( \Sigma \)-algebra of
Beh(SP) such that \( A = \text{Sig} \ T_{SP} \), then for every algebra \( B \) of Beh(SP) there is a unique \( \text{Sig} \)-behaviour morphism \( f : A \rightarrow B \).

2.2 Parameterized behaviour specifications

A parameterized specification is, as in the standard case, a pair of behaviour specifications. It will be required for the formal parameter to have all its sorts as observable sorts. This condition is technically necessary for the proper working of parameter passing. Moreover, it is also suitable from the methodological point of view because it is reasonable to consider that the result of the parameterization is observed through the parameter. In fact, all the authors working in the behaviour approach assume this kind of treatment [Gan 83, EhKr 82, GoMe 82, SaWi 82].

There exist free functors in the category of behaviours. Semantics of a parameterized specification is formed by all free functors behaviourally equivalent to the usual free functor. Persistency of those free functors can be characterized in the usual way in terms of consistency and sufficient conditions.

A morphism between two behaviour specifications is defined as in the standard case, but explicitly determining the relationship between the observable sorts. For a given morphism \( h : SP1 \rightarrow SP2 \), the relation that seems to be the more suitable from the methodological point of view is established by the requirement \( h(\text{Obs}1) \subseteq \text{Obs}2 \), that is, \( h \) has to translate observable sorts to observable sorts. We could also have required \( h(\text{S1-Obs}1) \subseteq \text{S2-Obs}2 \), that is, \( h \) would translate observable sorts to observable sorts and non observable sorts to non observable sorts. Nevertheless, we have not assume this additional condition because it does not present any technical disadvantage and it carries more generality.

**Definition 2.2.1**

Let \( SP1 = (\text{Obs}1, S1, \Sigma1, E1) \) and \( SP2 = (\text{Obs}2, S2, \Sigma2, E2) \) be two specifications.

A behaviour specification morphism \( h : SP1 \rightarrow SP2 \) is a specification morphism such that \( h(\text{Obs}1) \subseteq \text{Obs}2 \).

The forgetful functor associated to \( h \), \( U_h : \text{Beh}(SP2) \rightarrow \text{Beh}(SP1) \), is defined as usual. There is a free functor, \( \text{Free}_h \), left adjoint to the forgetful functor \( U_h \).

**Definition 2.2.2**

Let \( A \) be a \( \text{Sig} \)-álgebra.

**Theorem 2.2.3**

Let \( SP1 = (\text{Obs}1, S1, \Sigma1, E1) \) and \( SP2 = (\text{Obs}2, S2, \Sigma2, E2) \) be two specifications and
h : SP1 → SP2 a behaviour specification morphism. For every algebra A of Beh(SP1), the algebra

\[ \text{Free}_h(A) = T_{\Sigma^2}(A_{\text{Obs}1})/ =_{\text{obs}(A)} + \text{E2} \]

(interpreting the values \( a \in A_s, s \in \text{Obs}1 \), as values of sort \( h(s) \)) is a free construction over \( A \) with respect to \( U_h \) which extends to a free functor

\[ \text{Free}_h : \text{Beh}(SP1) \to \text{Beh}(SP2) \]

where the observable equations of \( A \) are

\[ \text{obs}(A) = \{ t_1 = t_2 \mid t_1, t_2 \in T_{\Sigma}(A_{\text{Obs}}), s \in \text{Obs} \gamma \varepsilon_A(t_1) = \varepsilon_A(t_2) \} \]

In what follows, if any possible confusion can arise, we will miss out any reference to the behaviour specification morphism over which the functor Free is defined.

**Definition 2.2.4**

Let \( \text{SPp} = (\text{Obsp}, \text{Sp}, \Sigma_p, \text{Ep}) \) and \( \text{SPc} = (\text{Obsc}, \text{Sc}, \Sigma_c, \text{Ec}) \) be two behaviour specificaciones. A parametrized behaviour specification \( \text{SPP} \) is a pair of behaviour specifications \( \text{SPP} = (\text{SPp}, \text{SPc}) \) such that \( \text{SPp} \subseteq \text{SPc} \). \( \text{SPp} \) is called the formal parameter specification and \( \text{SPc} \) body specification.

By reasons of simplicity in notation, we will consider sometimes that \( \text{Obsc} = \text{Obsp} + S_0, \text{Sc} = \text{Sp} + S_1 \) with \( S_0 \subseteq S_1, \Sigma_c = \Sigma_p + \Sigma_1 \) and \( \text{Ec} = \text{Ep} + \text{E1} \) where + stands for disjoint union. We will also indicate by \( \text{SPc}(\text{SPp}) \) the parametrized specification \( \text{SPP} \).

The semantics of a parametrized behaviour specification is defined as the set of all those constructions behaviourally equivalent to this Free construction. Due to that in these categories isomorphy is behaviourally equivalence, any construction behaviourally equivalent to a free construction is also free. Let us notice that, due to all the parameter sorts are observable, Free coincides at the algebraic level with the standard construction.

**Definition 2.2.5**

The semantics of a parametrized behaviour specification \( \text{SPP} = (\text{SPp}, \text{SPc}) \) is the class of functors

\[ \{ F : \text{Beh}(\text{SPp}) \to \text{Beh}(\text{SPc}) \mid F \equiv \text{Free} \} \]

**2.3 Parameter passing**

In this section (non parametrized) parameter passing for parameterized behaviour specifications is
presented. It will be necessary, as in the standard case, to precise the relation between formal and actual parameter, the construction that gives the resultant specification of parameter passing and, finally, the conditions which guarantee the correct working of the whole mechanism.

In the standard case, the connection between formal and actual parameter is established by means of a specification morphism. Obviously, it will be necessary here to use a similar mechanism, but the specification morphisms defined above are not the right ones.

The first problem arises with the relation that specification morphisms establishes between the observable sorts. The image by a specification morphism of an observable sort is an observable sort. Although, this condition is assumed by other authors, it is absolutely unreal from a practical point of view. In effect, if we think in the parameterized specification "set sof elements" we expect not to have any problem defining the type "sets of naturals" because the sort "nat" would probably be considered as observable. Nevertheless, we would get into troubles if we try to have "sets of sets of naturals" because the sort "set" is non observable in the specification "sets of naturals" and we would not be allowed to associate an observable sort to a non observable one.

The second problem is connected with the relationship between the equations of the formal and actual parameter. It does not seem very reasonable to ask the actual specification for containing itself in the equations (translated by the morphism) of the formal parameter. It would be enough for these equations to be behaviourally deduced because that would be in accordance with the idea underlying in the behaviour specification.

Consistently we have defined a new kind of connection called parameter passing morphism, taking into account these two problems. However these constructions are not really morphisms because they are not closed by composition. But this fact does not create additional problems.

Due to the special nature of a parameter passing morphism, it is not possible to associate to it a forgetful functor in the usual way in order to have a semantic mechanism for the parameter passing. So a new transformation will be needed.

At the syntactical level, the construction of the result of the parameter passing is made following a similar technique to the standard case: a pseudo-pushout is constructed from the actual parameter and the body of the parameterization.

The condition that guarantees the correctness of the parameter passing is, also as in the standard case, behavioural consistency and behavioural sufficient completeness.

2.3.1 Parameter passing morphisms

Such as it has yet been explained, a parameter passing morphism has to associate an observable sort to a non observable one. Moreover, it has to guarantee the admissibility of the actual parameter, that is, that the actual parameter satisfies the conditions imposed by the formal parameter. That means that the properties of the formal parameter $h^\sim(E)$ are consistent with respect to the observable properties of the formal parameter. Thus an actual parameter has only to satisfy the requirements of the formal parameter in a behavioural sense.
Definition 2.3.1.1

Let SP and SP' be two behaviour specifications.

A parameter passing morphism \( h : SP \rightarrow SP' \) is a signature morphism such that \( h(S\text{-Obs}) \subseteq S'\text{-Obs} \) and \( SP' \supseteq h^\sim(E) \).

Definition 2.3.1.2

Let \( h : SP \rightarrow SP' \) be a parameter passing morphism and \( A' \) an algebra of \( \text{Beh}(SP') \). The view realization of \( A' \) is the algebra

\[
\text{View}(A') = T_{\Sigma'}(A'Obs') / \equiv_{\text{obs}(A') + h^\sim(E)}
\]

Proposition 2.3.1.3

The view realization of \( A' \) is behaviourally equivalent to \( A' \).

Definition 2.3.1.4

Given a parameter passing morphism \( h : SP \rightarrow SP' \), we indicate by \( E'* + h^\sim(E) \) the specification \((S', S', \Sigma', E'* + h^\sim(E))\) which is called the view specification of \( SP' \) through \( SP \) and \( h \).

We will call \( \text{ViewRealiz}(SP') \) the full subcategory of \( \text{Alg}(V(SP,SP',h)) \) of all those algebras which are view realizations of algebras of \( \text{Beh}(SP') \).

We will call View functor to the functor \( \text{Realiz}_{E'* + h^\sim(E)} \), view of \( SP' \) through \( E'* + h^\sim(E) \), which transforms algebras of \( \text{Beh}(SP') \) to algebras of \( \text{ViewRealiz}(SP') \).

The functor \( V_h : \text{Beh}(SP') \rightarrow \text{Beh}(SP) \) defined by \( V_h = U_h \). View is called change of visibility functor associated to \( h \).

2.3.2 Syntax and semantics. Correctness

With the concepts defined in the previous section we are now able to define the semantics of the parameter passing for behaviour parameterized specifications.

Definition 2.3.2.1

Let \( \text{SPc}(SPp) \) be a behaviour parameterized with \( SPp = (\text{Obsp}, Sp, \Sigma p, Ep) \) and \( \text{SPc} = (\text{Obsc}, Sc, \Sigma c, Ec) = SPp + (S0, S1, \Sigma 1, E1) \). Let SPa be a behaviour and \( h : SPp \rightarrow SPa \) parameter passing morphism.

The instantiation of \( \text{SPc}(SPp) \) with the actual parameter \( SPa \) through \( h \) is

1. Sintactically : the specification \( SPr = (\text{Obsr}, Sr, \Sigma r, Er) \) defined by

\[
\begin{align*}
\text{Obsr} & = \text{Obsa} + S0 \\
\Sigma r & = \Sigma a + h'(\Sigma 1) \\
\text{Sr} & = \text{Sa} + S1 \\
\text{Er} & = \text{Ea} + h^\sim(\text{Ec})
\end{align*}
\]
where h' is a parameter passing morphism from SPc to SPr which extends h in the form:
   a) h'_{gen}(s) = if s ∈ Sp then h(s) else s
   b) h'_{ops}(σ) = if σ : s_1...s_n → s_{n+1} ∈ Σp then h(σ) else σ : h'(s_1)....h'(s_n) → h'(s_{n+1})

We will indicated by SPc(SPa)^h to the instantiation SPr. A instantiation determines a diagram called parameter passing diagram

```

```

where i_1 and i_2 are the behaviour morphisms associated to the inclusions of the specifications.

2. Semanticaly : the pair of functors
   { Free : Beh(SPp) → Beh(SPc) , Free' : Beh(SPa) → Beh(SPr) }

Analogous instantiations and diagrams would be defined for any injective behaviour morphisms i_1 and i_2.

Definition 2.3.2.2

Let SPc(SPp) be a parameterized behaviour specification and SPr and instantiation through the parameter passing morphism h : SPp → SPa. The semantics of the parameter passing is the pair of functors

```

```

```

```

{ Free : Beh(SPp) → Beh(SPc) , Free' : Beh(SPa) → Beh(SPr) }

Definition 2.3.2.3

A parameter passing is correct if:

1. For every algebra A of Beh(SPa) it holds that A ≅_{Σa} U_{i_2}·Free'(A). This property is called actual parameter protection.

2. It holds the functor equivalence Free·V_h ≅ V_{h'}·Free'. This property is called parameter passing compatibility.

Now we are going to determine under which conditions a parameter passing is correct. It will be necessary to ask for persistency in the source parameterized specification.
Theorem 2.3.2.4

Let Spc(SPp) be a parameterized behaviour specification with Obsp = Sp. Let SPa be the specification of an admissible actual parameter, h the corresponding parameter passing morphism and SPr(SPa) the result specification. If Spc(SPp) is behavioural consistent and sufficient complete then SPr(SPa) is behavioural consistent and sufficient complete.

Theorem 2.3.2.5

Let SPP = (SPp, Spc) be a parameterized behaviour specification. Parameter passing is correct for SPP and for every parameter passing morphism h : SPp → SPa if and only if the specification SPP is behavioural consistent and sufficient complete.

3. Implementations

The concept of implementation is strongly bound to the behaviour concept. Exactly, the simulation relationship between specifications which was commented in the previous section can be expressed in this context. We could say at a very intuitive level that a specification SP1 simulates a specification SP2 if, taking away all the stuff of SP1 not belonging to SP2 (sorts and operations), both behaviours are equal.

This intuitive idea is going to be formalized in this section by means of the concept of implementation morphism. An implementation morphism associates to the implemented or simulated specification the implementing specification. Those are a kind of morphisms different from the ones we have considered until now. They can be seen as specification morphisms with a restriction similar to persistency. This restriction assures the correctness of the "simulation".

Nevertheless, this kind of restriction can be considered too strong because of the over specification to which would lead the protection of the observable parts of the implemented specification. Although we agree with this criticism, it is really difficult to try to solve this problem without the aid of partial operations. In this work we have preferred to focus ourselves first on the treatment of total algebras and, consistently, this limitation is a technical necessity. In fact, a lot of works on implementation assume similar restrictions [BMPW 85, Ech 82, EhKr 82, EKMP 82, Jan 83, GoMe 82, GTW 78, Hoa 72, Ore 81, Ore 85, SaTa 87, SaWi 82].

We believe that this concept is essentially more simple and natural than the classical one [EKMP 82]. It can be seen in the framework of behavioural semantics as a special case of the institution independent concept of implementation called in [SaTa 87] "abstractor implementations".
3.1 Implementation morphisms

Given any quadruple $C = (\text{Obs}, S, \Sigma, E)$ of observable sorts, sorts, operations and equations, we will indicate by $\text{obs}(C)$, $\text{gens}(C)$, $\text{ops}(C)$ and $\text{ecns}(C)$ to $\text{Obs}$, $S$, $\Sigma$ and $E$ respectively. We will also write $\text{ctes}(C)$ to refer to the set of constant operations of $\text{ops}(C)$.

Definition 3.1.1

Let $\text{SPP}_1 = (\text{SP}_p, \text{SP}_p + \text{SP}_1)$ and $\text{SPP}_2 = (\text{SP}_p, \text{SP}_p + \text{SP}_2)$ be two parameterized behaviour specifications and $\text{mi} : \text{SPP}_1 \rightarrow \text{SPP}_2$ a signature morphism. We will say that $\text{mi}$ is an implementation morphism if

1. $\text{mi} \mid \text{SP}_p = \text{Id} \mid \text{SP}_p$

2. $\text{mi}(\text{obs}(\text{SPP}_1)) \subseteq \text{obs}(\text{SPP}_2)$

3. for every pair of terms $t_1$ and $t_2$ of $\text{T}_{\text{ops}}(\text{SPP}_1)(\text{X}_{\text{obs}(\text{SPP}_1)})$, $s \in \text{obs}(\text{SPP}_1)$ it holds

   $\text{ecns}(\text{SPP}_1) \vdash t_1 = t_2 \iff \text{ecns}(\text{SPP}_2) \vdash \text{mi}(t_1) = \text{mi}(t_2)$

4. for every term $t$ of $\text{T}_{\text{ops}}(\text{SPP}_2)(\text{X}_{\text{mi}(\text{obs}(\text{SPP}_1)})$, with $s \in \text{mi}(\text{obs}(\text{SPP}_1))$, there exists a term $t'$ of $\text{T}_{\text{ops}}(\text{SPP}_1)(\text{X}_{\text{obs}(\text{SPP}_1)})$ such that

   $\text{ecns}(\text{SPP}_2) \vdash \text{mi}(t') = t$

Associated to every implementation morphism there is always a sort of forgetful functor.

Proposition 3.1.2

Let $\text{mi} : \text{SPP}_1 \rightarrow \text{SPP}_2$ be an implementation morphism. The forgetful functor $U_{\text{mi}}$ associated to the signature morphism $\text{mi}$ is a functor from $\text{Beh}(\text{SPP}_2)$ to $\text{Beh}(\text{SPP}_1)$.

Definition 3.1.3

Let $\text{SPP}_1 = (\text{SP}_p, \text{SP}_p + \text{SP}_1)$ and $\text{SPP}_2 = (\text{SP}_p, \text{SP}_p + \text{SP}_2)$ be two parameterized behaviour specifications. $\text{SPP}_2$ implements to $\text{SPP}_1$ if there exists an implementation morphism $\text{mi} : \text{SPP}_1 \rightarrow \text{SPP}_2$. 

10
An implementation can be seen in a certain way as a dual mechanism of the instantiation mechanism. To a non observable sort it can be associated an observable one by means an implementation morphism \( m_i : SPP1 \to SPP2 \). Specifically, the implementation of non observable objects can be done, and so will be in a lot of cases, in terms of observable sorts which become hidden objects just in the moment when the implementation is realized. For instance, a stack can be implemented using an array and a pointer which are probably basic types of a programming language and consistently can be considered as observable. Nevertheless, since the moment the implementation is done, these objects would remain hidden. In this way the passage from SPP2 to SPP1 reflects the hiding of the implementation. Definitively, the unique sorts really observable are those corresponding to the basic types implemented in the programming environment and which consistently will set up the behaviour of the specifications defined over them.

The third given condition for a signature morphism to be an implementation morphism means that all the observable relations derived from SPP1 are also derived from SPP2 and that in SPP2 no observable relations are added to the behaviour of SPP1. It could be said that this condition expresses a kind of observable consistency property. In this way the implementation of SPP1 really implements the behaviour of SPP1. This condition together with the fourth one, which reflects a sort of observable sufficient completeness property, assure the protection of the observable part. It can be thought that both conditions are too strong, nevertheless it is quite natural that the observable part remains protected. Perhaps a more suitable form of handling this problem would be the use of partial operations over the representation domain.

However, what is asked for a signature morphism to be a parameter passing morphism is that the behaviour of the actual parameter fulfills the requirements established in the formal parameter. That means, in a certain sense, that the non observable properties and objects of the actual parameter can "become observable" during the parameter passing. Let us suppose, for instance, that the formal parameter defines an observable sort "value" and an "equal" operation with equations for reflexivity, symmetry and transitivity and that the passing morphism associates to "value" a non observable sort "value*" and to the operation "equal" a operation "equal*" in the actual parameter. This "equal*" operation perhaps does not directly satisfy the properties of "equal" but it will certainly satisfy them in a behaviour manner. In the parameter passing we turn "value*" as observable in order to verify that the "equal" operation is an equality behaviourally speaking.

\[ 1.2 \text{ Parameter passing and implementation} \]

The following proposition establishes the compatibility between instantiation and
implementation of parameterized behaviour specifications.

**Theorem 3.2.1**

Let $SPP2 = (SPp, SPp+SP2)$ be a parameterized behaviour specification which implements $SPP1 = (SPp, SPp+SP1)$ according to the implementation morphism $mi : SPP1 \rightarrow SPP2$. Let $SPP'$ be a parameterized behaviour specification and $h : SPp \rightarrow SPP'$ a behaviour specification morphism. Let $SPP3 = SPP1(SPP')^h$ be the instantiation of $SPP1$ with the actual parameter $SPP'$ through $h$ and $SPP4 = SPP2(SPP')^h$ the instantiation of $SPP2$ with the actual parameter $SPP'$ through $h$. It holds that $SPP4$ is an implementation of $SPP3$.

**Proof**

Let $h'$ be the behaviour specification morphism extension of for the actualization of $SPP1$ with $SPP'$, that is, commutes the parameter passing diagram

\[
\begin{array}{ccc}
SPp & \rightarrow & SPP1 = (SPp, SPp+SP1) \\
\downarrow h & & \downarrow h' \\
SPP' & \rightarrow & SPP1(SPP')^h \\
\end{array}
\]

Let $h''$ be the behaviour specification morphism extension of for the actualization of $SPP2$ with $SPP'$, that is, commutes the parameter passing diagram

\[
\begin{array}{ccc}
SPp & \rightarrow & SPP2 = (SPp, SPp+SP2) \\
\downarrow h & & \downarrow h'' \\
SPP' & \rightarrow & SPP2(SPP')^h \\
\end{array}
\]

We have to define a signature morphism $mi'$ which commutes the diagram

\[
\begin{array}{ccc}
SPP1 = (SPp, SPp+SP1) & \rightarrow^mi & SPP2 = (SPp, SPp+SP2) \\
\downarrow h' & & \downarrow h'' \\
SPP3 = (SPP', SPP'+h'(SP1)) & \rightarrow^{mi'} & SPP2(SPP')^h \\
\end{array}
\]

12
We define

\[ mi'_{\text{gen}}(s) = \begin{cases} 
& \text{if } s \in \text{gens}(SPP') \text{ then } s \\
& \text{else } mi(s) 
\end{cases} \]

\[ mi'_{\text{ops}}(\sigma : s_1 \cdots s_n \rightarrow s) = \begin{cases} 
& \text{if } \sigma \in \text{ops}(SPP') \text{ then } \sigma \\
& \text{else } h'' \cdot mi(h''^{-1}(\sigma)) 
\end{cases} \]

It holds that \( mi' \) defined in this way commutes the previous diagram and therefore also commutes the following diagram:

\[
\begin{align*}
&\text{SP}_p \\
\downarrow \text{mi} &\quad \downarrow h &\quad \downarrow h'' \\
\text{SPP}_1 = (\text{SP}_p, \text{SP}_p + \text{SP}_1) &\quad \text{SPP}_2 = (\text{SP}_p, \text{SP}_p + \text{SP}_2) \\
\downarrow h' &\quad \downarrow h &\quad \downarrow h'' \\
\text{SPP'} &\quad \text{SPP'} \\
\downarrow \text{mi'} &\quad \downarrow \text{mi'} \\
\text{SPP}_1(\text{SPP'}) &\quad \text{SPP}_2(\text{SPP'}) \\
\end{align*}
\]

Moreover is an implementation morphism as is deduced from the following lemmas.

**Notation**

The morphism \( mi' \) will be denoted by \( mi'^h \).

**Lemma 3.2.2**

Let \( \text{SPP}_1, \text{SPP}_3, h'' \) and \( mi' \) as in the theorem 3.2.1.

Let \( z \) be a variable such that \( \text{gen}(z) \in h'(\text{gens}(\text{SPP}_1)) \).

There exist a mapping

\[
\rho : \text{mi'}(\text{ops}(\text{SPP}_3)(X_{\text{obs}}(\text{SPP}_3) \cup \{z\}))_{\text{obs}}(\text{SPP}_3) \rightarrow \\
\rightarrow h'' \cdot \text{mi'}(\text{ops}(\text{SPP}_1)(X_{\text{gens}}(\text{SPP}_1) \cup \{z'\}))_{\text{obs}}(\text{SPP}_1)
\]

and an assignment

\[
\sigma : h'' \cdot \text{mi'}(X_{\text{gens}}(\text{SPP}_1) \cup \{z'\}) \rightarrow \text{mi'}(\text{ops}(\text{SPP}_3)(X_{\text{obs}}(\text{SPP}_3) \cup \{z\}))
\]
such that $\bar{\sigma}_p(\rho(c[z])) < c[z]$ para todo $c[z]$ de $\text{mi}'(\text{T}_{\text{ops}}(\text{SPP}3))\langle X_{\text{obs}}(\text{SPP}3) \cup \{z\} \rangle |_{\text{obs}}(\text{SPP}3)$.

Lemma 3.2.3

There exist a mapping

$\tau : \text{mi}'(\text{T}_{\text{ops}}(\text{SPP}3))\langle X_{\text{obs}}(\text{SPP}3) \rangle |_{\text{h''}} \cdot \text{mi}'(\text{gens}(\text{SPP}1)) \rightarrow \text{h''} \cdot \text{mi}'(\text{T}_{\text{ops}}(\text{SPP}1))\langle X_{\text{obs}}(\text{SPP}1) \rangle$

and an assignment

$\sigma_\tau : \text{h''} \cdot \text{mi}'(X_{\text{obs}}(\text{SPP}1)) \rightarrow \text{mi}'(\text{T}_{\text{ops}}(\text{SPP}3))\langle X_{\text{obs}}(\text{SPP}3) \rangle$

such that $\bar{\sigma}_\tau(\tau(t)) = t$.

Lemma 3.2.4

Let SPP1, SPP3 $h'$, $h''$ and mi' be as in the theorem 3.2.1. Let $S^\#$ be the set defined by $S^\# = \text{obs}(\text{SPP}3) \cup h'(\text{obs}(\text{SPP}1))$.

There exists a mapping

$f_{\text{mi}'} : \text{T}_{\text{ops}}(\text{SPP}4)(X_{\text{mi}'}(S^\#)) |_{\text{mi}'}(S^\#) \rightarrow \text{T}_{\text{ops}}(\text{SPP}3)(X_{S^\#}) |_{S^\#}$

such that $h''(\text{ecns}(\text{SPP}2)) \vdash \text{mi}'(f_{\text{mi}'}(t)) = t$.

Remarks 3.2.5

Due to that the mapping $f_{\text{mi}'}$ satisfies properties 1. and 2. mentioned at the beginning of the previous proof and due to the way $f_{\text{mi}'}$ has been defined in lemma 3.2.4, the mapping $f_{\text{mi}'}$ satisfies:

1. if $t \in \text{T}_{\text{ops}}(\text{SPP}3)(X_{S^\#})$ then $\text{SPP}3 \vdash f_{\text{mi}'}(\text{mi}'(t)) = t$

2. if $t \in \text{T}_{\text{ops}}(\text{SPP}4)(X_{\text{mi}'}(S^\#))$ and $\alpha \in \text{Pos}(t)$ such that $\text{gen}(\langle \alpha \rangle) \in \text{mi}'(S^\#)$ then

$h'(\text{ecns}(\text{SPP}1)) \vdash f_{\text{mi}'}(t[\alpha \leftarrow x])([x \leftarrow (f_{\text{mi}'} \cdot \bar{\sigma})(t')]) = f_{\text{mi}'}(t[\alpha \leftarrow \bar{\sigma}(t')])$

for every $t'$ in $\text{T}_{\text{ops}}(\text{SPP}4)(X_{\text{mi}'}(S^\#))$ with sort in $\text{mi}'(S^\#)$ and for every assignment

$\sigma : X_{\text{mi}'}(S^\#) \rightarrow \text{T}_{\text{ops}}(\text{SPP}4)(X_{\text{mi}'}(S^\#))$
such that $t_{\mathbf{x}} = \overline{\sigma(t')}$. 

**Lemma 3.2.6**

The signature morphism $m'$ defined in theorem 3.2.1 is an implementation morphism.

**4. Behaviour modules**

As it has yet been mentioned above, a behaviour module $BM$ consists of an interface part, an implementation part, and enrichment and an implementation morphism. The interface part is the export specification of the behaviour module $BM$, that is, gives what $BM$ offers to users. The implementation part is another behaviour module and constitutes the import part of $BM$ which a lot of times has to be enriched with other sorts and/or operations in order to achieve the whole implementation of the interface. The implementation morphism takes care of the relationship between the interface of $BM$ and the interface of the implementing behaviour module plus the enrichment. So, in this way, a behaviour module can be seen as a "waterfall" of implementations which leads from the interface implementation to the predefined types of the specification environment which are yet directly implemented.

**Definition 4.1**

A **behaviour module** $BM$ is a 4-tuple $BM = (SPint, BM', SPenr, im)$ where $SPint$ is a persistent parameterized behaviour specification called interface specification and $SPenr$ is a persistent parameterized behaviour specifications enrichment specification and designed by interface($BM$) and enrich($BM$) respectively. $BM'$ is another behaviour module called implementation of the behaviour module $BM$ and designed by impl($BM$). Moreover, $SPenr$ is a conservative extension of interface($BM'$) and $im: SPint \rightarrow$ interface($BM'$) + $SPenr$ is an implementation morphism denoted by impl$_{\text{morph}}$(BM). A behaviour module $BM$ is called predefined if is of the form $BM = (SPint, \emptyset, \emptyset, \text{null})$. 
At the semantical level a behaviour module has the same meaning as its interface, that is, a free functor up to behavioural equivalence. The reason is that this functor corresponds to the meaning of the sole part of the behaviour module which is really visible because the implementation part is completely inaccessible.

**Definition 4.2**

Let $BM = (SP\text{Int}, BM', SP\text{Enr}, \text{im})$ be a behaviour module with $SP\text{Int} = (SP\text{Int}_p, SP\text{Int}_b)$. The behavioural semantics of $BM$ is $\text{Sem}(SP\text{Int})$, that is

$$\text{Sem}(BM) = \{ F : \text{Beh}(SP\text{Int}_p) \to \text{Beh}(SP\text{Int}_b) \mid F \text{ is free} \}$$

Due to the composition of a behaviour module $BM_0 = (SP\text{Int}_0, BM_1, SP\text{Enr}_0, \text{im}_0)$, a functor $F_0$ in $\text{Sem}(BM_0)$ can be split up into a composition of free functors which reflects the construction of the total implementation starting from the predefined types of the environment. Specifically, the functor $F_0$ is behaviourally equivalent to

$$F_0 \equiv V_{\text{im}_0} F_{\text{Enr}_0} F_1$$

where $V_{\text{im}_0}$ is the forgetful functor associated to the implementation morphism $\text{mi}_0$ and $F_{\text{Enr}_0}$ is the functor associated to the free enrichment of the specification interface $(BM_1)$ with the specification $SP\text{Enr}_0$.

![Diagram](attachment:diagram.png)

So, if we have

$$U_j = (SP\text{Int}_j, BM(j+1), SP\text{Enr}_j, \text{im}_j) \text{ for every } j \in \{1, \ldots, r-1\}$$

where $BM_r$ is a behaviour module with a predefined specification as interface. Let $F_j$ be the functors corresponding to the semantics of $BM_j$ for every $j \in \{1, \ldots, r-1\}$. It holds that

$$F_j \equiv V_{\text{im}_j} F_{\text{Enr}_j} F(j+1) \text{ for } j \in \{0, \ldots, r-1\}$$

and consistently $F_0$ can be decomposed in the form
\( F_0 \equiv V_{im0} \cdot F_{0\text{enr}} \cdot V_{im1} \cdot F_{1\text{enr}} \cdot \ldots \cdot V_{im(r-1)} \cdot F_{(r-1)\text{enr}} \cdot F_r \)

This new expression for the semantics of BM0 explicitly gives the complete implementation of the interface specification of BM0. It can be said that \( F_0 \) is the "interface functor" and the right side of the above equivalence can be called the "realization functor" of BM0.

### 4.1 Combination of behaviour modules

Behaviour modules can be combined to get new behaviour modules. The basic operations allowed to combine modules are the enrichment, the disjoint union and the actualization.

**Definition 4.1.1**

Let \( BM_j = (SP_{j\text{int}}, BM_j', SP_{j\text{enr}}, im_j), j \in \{1,\ldots,n\} \) be \( n \) behaviour modules. The behaviour module BM is the disjoint union of \( BM_1, \ldots, BM_n \), denoted by \( BM_1 + \ldots + BM_n \), if is of the form:

\[
\text{interface}(BM) = \text{interface}(BM_1) + \ldots + \text{interface}(BM_n)
\]

\[
\text{impl}(BM) = \begin{cases} \text{impl}(BM_j) & \text{if predefined?}(BM_j) \\ \text{impl}(BM_1) + BM_2 + \ldots + BM_n & \text{else} \end{cases}
\]

\[
\text{enrich}(BM) = \begin{cases} \text{enrich}(BM_1) & \text{if predefined?}(BM_1) \\ \text{interface}(BM_1) & \text{else} \end{cases}
\]

\[
\text{impl\_morph}(BM) = \begin{cases} \text{id} & \text{if predefined?}(BM_1) \\ \text{impl\_morph}(BM_1) & \text{else} \end{cases}
\]

where

\[
\text{impl\_morph}(BM_1) : \text{interface}(BM) \rightarrow SP^#
\]

with

\[
SP^# = \text{interface}(\text{impl}(BM_1)) + \text{enrich}(BM_1) + \text{interface}(BM_2 + \ldots + BM_n)
\]

is defined on \( \text{interface}(BM_1) \) as \( \text{impl\_morph}(BM_1) \) and on the rest as the identity, that is

\[
\text{impl\_morph}(BM_1)^# | \text{interface}(BM_1) = \text{impl\_morph}(BM_1)
\]

\[
\text{impl\_morph}(BM_1)^# | \text{interface}(BM_2 + \ldots + \text{interface}(BM_n)) = \text{id}
\]
This construction defines a behaviour module which is the union of some given behaviour modules. The specification of the union interface is the union of interfaces. If the first one BM1 is predefined then it is removed from the set of modules to be joined later and is directly joined to the enrichment of the implementation of the result. Otherwise the union of BM2, ..., BMn and impl(BM1) is performed.

If, for instance, BM1 is a module defining the sets of values and BM2 is the stacks of elements

![Diagram of BM1 and BM2](image)

then the module which is the union of both of them is defined in the form

![Diagram of BM1 + BM2](image)

The resulting module BM, disjoint union of BM1, ..., BMn, is a well defined module, that is, the signature morphism impl_morph(BM) is an implementation morphism.

**Proposition 4.1.2**

The module BM defines as the union of BM1, ..., BMn is a correct behaviour module.

The enrichment of behaviour modules BM1, ..., BMn with the sorts, operations and equations of a 4-tuple C is a module BM which has as interface the union of the specifications
interface(BM1), ..., interface(BMn) and then enriched with C. If in C there are new sorts and operations there will be necessary to give their implementation in order to obtain a well defined module, that is, completely implemented. In the definition of that implementation would be necessary an auxiliary module BM'. If this module BM' does not appear, then is BM' = (Ø, Ø, Ø, null), we will say that we have a null module. In this case all the new things of C is perfectly implemented in terms of BM1, ..., BMn. That is the situation corresponding to a new operation which is a composition (derived operation) of operations of BM1, ..., BMn, so its implementation is the set of equations of C which define this operation. Therefore, the implementation module is just the union of BM1, ..., BMn.

The enrichment of the implementation will be the enrichment of interface(impl(BM)) with (Ø, Ø, {σ₁}_{i ∈ {1..r}}, {e₁}_{i ∈ {1..s}}), which, as usual, should be a conservative extension.

Definition 4.1.3

Let BMj = (SPjint, BMj', SPjentr, imj), j ∈ {1,...,n} be n behaviour modules, BM' another behaviour module, C a 4-tuple of observable sorts, sorts, operations and equations, {σ₁}_{i ∈ {1..r}} a set of operations, {e₁}_{i ∈ {1..s}} a set of equations and h a signature morphism from interface(BM1) + ... + interface(BMn)+C to interface(BM')+(Ø, Ø, {σ₁}_{i ∈ {1..r}}, {e₁}_{i ∈ {1..s}}) in the case where BM' is not null, or from interface(BM1) + ... + interface(BMn)+C to interface(BM1) + ... + interface(BMn)+(Ø, Ø, {σ₁}_{i ∈ {1..r}}, {e₁}_{i ∈ {1..s}}) with h |interface(BM1) + ... + interface(BMn) = id |interface(BM1) + ... + interface(BMn) if BM' is not null.

The module BM enrichment of BM1, ..., BMn with C is defined by

interface(BM) = interface(BM1) + ... + interface(BMn) + C

impl(BM) = if null?(BM') then union(BM1, ..., BMn)
else BM'

enrich(BM) = interface(impl(BM)) + (Ø, Ø, {σ₁}_{i ∈ {1..r}}, {e₁}_{i ∈ {1..s}})

impl_morph(BM) = h

For instance, let us consider that we can dispose of the module INTEGER of integer numbers and the module STACK of stacks of elements implemented by the module ARRAY_POINTER of the tuples of array of elements and a pointer to determine the position in the array of the top of the represented stack. We may wish to enrich the stacks with an operation for computing the length of a stack. Then we define a module BM = STACK_with_LENGTH as an enrichment of the modules BM1 = INTEGER and BM2 = STACK with C = (Ø, Ø, {length: stack → integer}, {length(empty) = 0, length(push(p, x)) = suc(length(p)), length(pop(p)) = pred(length(p))} ), BM' = INTEGER - ARRAY_POINTER, the set {σ₁}_{i ∈ {1..r}} reduced to an unique operation
length': tuple → integer, the set \( \{e_i\}_{i \in \{1..s\}} \) having the three equations \( \{\text{length}'(\text{empty'}) = 0, \text{length}'(\text{push}'(p, x)) = \text{suc}(\text{length}'(p)), \text{length}'(\text{pop}'(p)) = \text{pred}(\text{length}'(p))\} \) and the renaming morphism \( h \) defined by \( h(\text{length}) = \text{length}' \). It is supposed that tuple is the sort of the interface of the module \text{ARRAY\_POINTER} and \text{empty}', \text{push}', \text{pop}' are the operations of \text{ARRAY\_POINTER} that implement the corresponding operations of the module \text{STACK}.

**Proposition 4.1.4**

The behaviour module \( \text{BM} \) defined in 4.1.3 is a correct behaviour module.

**Definition 4.1.5**

Let \( \text{BM}_1 = (\text{SP1int}, \text{BM}_1', \text{SP1enr}, \text{im}_1) \) and \( \text{BM}_2 = (\text{SP2int}, \text{BM}_2', \text{SP2enr}, \text{im}_2) \) be two behaviour modules and \( h: \text{interface}(\text{BM}_1) \rightarrow \text{interface}(\text{BM}_2) \) a parameter passing morphism.

The **actualization** of the behaviour module \( \text{BM}_1 \) with \( \text{BM}_2 \) through \( h \), denoted by \( \text{BM}_1(\text{BM}_2)^h \), is defined by

\[
\text{interface}(\text{BM}) = \text{interface}(\text{BM}_1)(\text{interface}(\text{BM}_2))^h
\]

\[
\text{impl}(\text{BM}) = \begin{cases} \text{if predefined?(BM}_1) & \text{then BM}_2 \\ \text{else} & \text{impl}(\text{BM}_1)(\text{BM}_2)^h \end{cases}
\]

\[
\text{enrich}(\text{BM}) = \begin{cases} \text{if predefined?(BM}_1) & \text{then interface(BM}_1) \\ \text{else} & h(\text{enrich(BM}_1)) \end{cases}
\]

\[
\text{impl\_morph}(\text{BM}) = \begin{cases} \text{if predefined?(BM}_1) & \text{then id} \\ \text{else} & \text{impl\_morph(BM}_1)^h \end{cases}
\]

The interface of the actualization of \( \text{BM}_1 \) with \( \text{BM}_2 \) through \( h \) is the actualization of the specifications of the interface of \( \text{BM}_1 \) with the interface of \( \text{BM}_2 \) through the parameter passing morphism \( h \). The module implementation of the actualization is obtained by successive actualizations starting in the actualization of \( \text{impl}(\text{BM}_1) \) with \( \text{BM}_2 \). This means that the interface of \( \text{BM} \) is the actualization of \( \text{impl}(\text{BM}_1) \) with \( \text{interface}(\text{BM}_2) \) and so on. This sequence of actualizations will finish when arriving to the actualization of a predefined module with \( \text{BM}_2 \). In this moment the module \( \text{BM}_2 \) can be **hung**.

The specification \( \text{enrich}(\text{BM}) \) is simply the renaming of the enrichment of \( \text{BM}_1 \) by the parameter passing morphism \( h \).

The new implementation morphism is \( \text{impl\_morph}(\text{BM}_1)^h \), that is, the extension of the implementation morphism of \( \text{BM}_1 \) by the parameter passing morphism \( h \) defined in 3.2.1.

For example, if \( \text{BM}_1 \) is a module for the stacks of elements and \( \text{BM}_2 \) is a module for the sets of values, as shown in the following figure.
the actualization module $BM_1(BM_2)^h$ has the following structure

**Proposition 4.1.6**

The behaviour module $BM = BM_1(BM_2)^h$ defined as the actualization of $BM_1$ with $BM_2$ through the parameter passing morphism $h$ is a correct behaviour module.

**5. Conclusions**

In order to specify in the large it is necessary to make use of a specification language with suitable mechanisms which facilitate construction, understanding, modification and reusability of specifications.

Since Clear [BuGo 80, BuGo 81] some specification construction operations have been considered as basic. Those are enrichments, (allowing to add sorts, operations and equations to an existing specification, and therefore allowing to construct specifications in a hierarchical fashion), renaming (allowing to obtain a new specification changing sort names and operation names in a
existing specification), unions (for combining some specifications), actualizations (allowing to reuse generic specifications by substitution of formal parameters by actual parameters) and hiding (allowing to reuse a previous specification removing all the stuff which is not necessary)

The early algebraic specification languages were OBJ [GJFM 85, GoMe 82a] and CLEAR [BuGo 77] of which in [BuGo 80] its denotational semantics was given. Some mistakes were later detected and revised in [San 81]. With the aim of to simplify CLEAR complexity the language LOOK [ETLZ 84] appeared with a definition which joints denotational semantics [ScSt 71] and algebraic semantics concepts. The OBJ language is founded in initial semantics and both LOOK and CLEAR are defined in the ambit of loose semantics with initial restrictions.

With strong influences of LOOK the language ACT ONE [EhMa 85] was defined in the framework of initial semantics with two levels of semantics, one for the syntactic aspect of specifications and the other for the algebraic aspect.

All these languages have either loose semantics with mechanisms allowing to restrict the class of models or initial semantics. For instance, the way of selecting models in CLEAR is the "data constraint" mechanism which forces to interpret some parts of a specification in a initial way.

The sole specification language which, as far as we know, explicitly mentions the behavioural notion is ASL [SaWi 83, Wir 86, AsWi 86]. This language has loose semantics but one of the constructions it has to delimit the class of "interesting" models of a specification is a behaviour abstraction operation.

One of the aims of this paper to give the bases for defining the full behavioural semantics of a specification language called MERLÍN-T. The two level behavioural semantics of this language can be found in [Ni 87]. MERLÍN-T can be considered as a specification/design language due to one of its constructions which corresponds to the module concept with a visible interface and a hidden realization. This construction, called "universe", is the basis for a language supporting description of program designs. Specifically, this language incorporates some operations for constructing and structuring universes.

Based on an implementation concept formalising the idea of simulating a given behaviour, we have defined in this paper a behaviour module concept which, due to its structure, can be used as a tool for the complete design of software systems. It has a visible interface establishing what the module offers to an user in a similar form as in programming languages (packages, clusters,...). The implementation part of a behaviour module, which is naturally hidden for an user, is formed by another behaviour module, which represents what is imported to be able to define the implementation of the interface, and an enrichment extending the imported module. The implementation relation between the interface and the implementation part is given by the
implementation morphism.

Behaviour modules can be interconnected by means of union, enrichment and actualization constructions yielding new correct behaviour modules. Some more topics are going to be studied as the general union of modules and the operation of hiding.

5. Bibliography


[ NiOr 87 ] Nivela, Mª P.; Orejas, F., Initial behaviour semantics for algebraic specifications, to appear in Proc. 5th Workshop on Algebraic Specification of Abstract Data Types, Gullane, 1.897, Lecture Notes in Computer Science.


[ Ore 85 ] Orejas, F., On implementability and computability in abstract data types, en


