Compositionality Results for Different Types of Parameterization and Parameter Passing in Specification Languages

H. Ehrig, Technical University Berlin
R. M. Jimenez, Universitat Politecnica de Catalunya
F. Orejas, Universitat Politecnica de Catalunya

1 Introduction

In the literature on specification several notions of parameterization have been studied. Some of these notions regard parameterizations only at the model level, i.e as functors or parameterized algebras that are used to build data types from other given data types in a well-specified manner. Examples in this sense are the pioneering approach of [TWW 82], where parameterized data types are defined as persistent free functors specified by inclusions of specifications, and the Extended-ML modules [ST 89]. A different approach consists in seeing parameterizations as parametric specification transformations. This view has been taken in [BG 80, Ehc 82], where parameterized specifications are defined as inclusions of specifications and where the transformation associated to parameter passing is "computed" by means of a pushout construction. A rather more general approach along this line can be found in [SW 83, Wir 86], where parameterizations are allowed to be arbitrary λ-expressions built over the ASL specification-building operations and parameter passing is defined as a form of β-reduction.

[EKTWW 84] provided the first synthesis of the two different lines by defining parameterizations both as specifications of parameterized data types and as parametric specification transformations in a compatible manner. However, the results of [EKTWW 84] applied only to the initial framework and to the case where parameterizations are defined as inclusions of specifications (i.e. [TWW 82] and [BG 80, Ehc 82, Li 83]).

Recently, in [SST 90], Sannella, Sokolowski and Tarlecki have studied the general case (i.e. parameterizations defined by arbitrary λ-expressions), within the loose framework, and have found that the two notions of parametric transformation of specifications (from now on, parameterized specifications) and specification of parameterized data types would not only be based on different intuitions, but would also be semantically different. Essentially, their point of view is that parameterized specifications are not proper specifications in the sense that they are not necessarily intended to describe a software unit but a specification transformation, i.e. their aim is to be useful at the specification design level, since they may be seen as user-defined specification-building operations. Conversely, specifications of parameterized data types are true specifications in the sense that are intended to describe (generic) software units. As a consequence the two notions denote different constructions: a parameterized specification denotes a function that maps specifications into specifications (or classes of models into classes of models), whereas a specification of parameterized data types describes a class of "parameterized programs", i.e. functions that map "programs" into "programs" (or models into models). To be more precise, we may consider that a "program" is a specification having a single model (up to isomorphism).

In this paper we go beyond [SST 90] in a number of ways. Firstly, and most important, we have defined the different notions of parameterization and parameter passing, both at the specification and at the model level, obtaining different compositionality results. On the other hand, all the results do not only apply to the loose approach but (because of the explicit handling of constraints) can also be directly applicable to the initial framework, and in general to any other kind of monomorphic framework. In particular, the results obtained can be used to generalize and extend all previous results for the initial approach (e.g. [TWW 82, BG 80, Ehc 82, EM 85]) to the stronger kind of parameterizations defined by arbitrary λ-expressions. Moreover, for obtaining all these results new categorical constructions of *multiple* pushouts, amalgamations and extensions had to be defined and studied.

The paper is organized as follows: in section two we provide an introduction to the framework of *specification frames* and to the new *multiple* categorical constructions, on which this paper is based. Section three describes the kind of operations allowed for building specifications. In sections 4 and 5 the different parameterization and parameter passing mechanisms are studied obtaining the main results. Finally, in section 6 some conclusions are drawn and some further work is sketched.

ACKNOWLEDGEMENTS

The authors would like to thank J. Goguen and A. Tarlecki for discussions and comments that have contributed to considerably improve the paper. Also, an anonymous referee provided some helpful comments. Finally, This work has been partially supported by ESPRIT Basic Research Working Group COMPASS (ref. 6112).

2 "Multiple" Constructions in Specification Frames

In this section we present the basic categorical constructions used in the paper. First, we introduce the notion of *specification frame* as an indexed category that satisfies some additional "structural" properties and, then, we present the new constructions that will be needed in this work. The reader is assumed to have some basic knowledge in category theory and algebraic specification. For more details on the specific contents of this paper, the reader may consult the full version and [BGT 91, EBCO 91, EM 85, EM 89, SST 90].

In this paper, the properties that we associate to specification frames are the existence of pushouts and amalgamations and foundedness. Our aim in using indexed categories (plus the additional properties) is to study different constructions for building and structuring specifications at a very abstract level, in such a way that the results obtained could be independent of the specific logic used for specification, as far as this logic satisfies the required properties, i.e.we can think that the results presented are "parameterized" by the the logic used for specification, with the current definition of specification frames being the formal parameter. In this sense, we do not consider the properties defining specification frames to be fixed but depending on the specific constructions studied, i.e. the term "specification frame" refers more to a catchword than to a concept.

Studying specification constructions at a very abstract level has now a long tradition in the framework of institutions [GB 84, GB 92]. In this context, we prefer to work using the simpler construction of indexed categories because our results are mainly based on the "structural" properties mentioned above whereas the notions of "formulae" and "satisfaction" only play an implicit role in our framework.

In some previous papers, [EPO 89, EBCO 91], we used the term "specification logic" instead of specification frame or indexed category because, as said above, our intention was to describe abstractly (some characteristics of) the logics used for specification. However, we have been convinced of the inadequacy of that name, taking into account that no notion of satisfaction or logical inference was explicitly involved.

2.1 Definition

A specification frame SF is an indexed category (**SPEC**, Catmod), where **SPEC** is a category of abstract specifications and Catmod: $\mathbf{SPEC}^{op} \rightarrow \mathbf{CATCAT}$ is a functor, that associates to every specification SP in **SPEC** its category of models Catmod(SP), such that the following properties are satisfied:

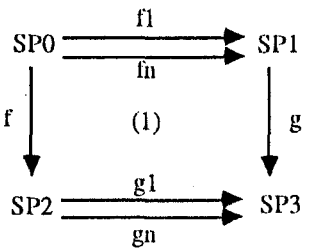
1. Pushouts: **SPEC** has pushouts.
2. Amalgamation: Catmod transforms pushouts into pullbacks.
3. Foundedness: **SPEC** has an initial object ØSP and Catmod(ØSP) = **1**, the category in **CATCAT** consisting of exactly one object and one morphism.

Most logics used for specification are specification frames. Pushouts are the operations that allow to combine specifications. Amalgamation is the semantic counterpart to pushouts. Finally, the empty specification and the "empty" model would be their ØSP, and Catmod(ØSP), respectively.

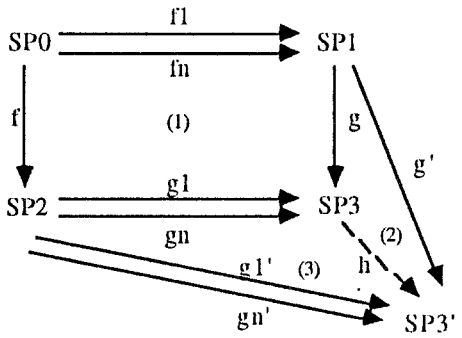
In "classical" parameter passing (see e.g. [EM 85]) pushouts, amalgamation and extension provided the basic tools for defining the various semantic concepts. For example, pushouts were used to describe parameter passing at the specification level, in the sense that the operation of substituting in the body of a parameterization the formal by the actual parameter specifications is a pushout in the category of specifications. However, in the general case studied in this paper, pushouts cannot be directly used to describe parameter passing in the same way. The problem is that the formal parameter may "occur" several times in the body specification and pushout only provide a "single" substitution. The construction needed, introduced below, is a *multiple pushout*.

2.2 Definition

Given morphisms f1,...,fn: SP0 → SP1 ($n \geq 0$) and f:SP0 → SP2 in **SPEC** the following diagram in **SPEC** is called multiple pushout of (f1,...,fn) and f if we have



1. (Graded Commutativity): $g \circ fi = gi \circ f \quad (i = 1,...,n)$
2. (Universal Property): For each object SP3' and morphisms g', g1,...,gn with $g' \circ fi = gi' \circ f$, $(i = 1,...,n)$ there is a unique morphism h s.t. $h \circ g = g'$ and $h \circ gi = gi'$ $(i = 1,...,n)$



Remarks

1. Note that SP3 is not the colimit object in diagram (1).

2. In the case n = 1 a multiple pushout is a pushout.

3. In the case n = 0 the multiple pushout object SP3 is equal to SP1 with $g = 1_{SP1}$.

As a consequence of the fact that finite coproducts can be constructed using pushouts and initial objects and of the fact that multiple pushouts can be constructed using finite coproducts and pushouts we have:

2.3 Proposition

If SF = (**SPEC**, Catmod) is a specification frame then **SPEC** has multiple pushouts.

In the same way that amalgamations are the semantic counter-parts of pushouts, we may define multiple amalgamation as the semantic construction associated to multiple pushouts.

2.4 Definition

An indexed category SF has multiple amalgamation if for every multiple pushout as above we have the following properties:

1. For all objects A1 in Catmod(SP1) and all families of objects $A2^* = (A2_1,...,A2_n)$ in Catmod(SP2) and $A0^* = (A0_1,...,A0_n)$ in Catmod(SP0) with $V_{fi}(A1) = A0_i = V_f(A2_i)$, for i = 1,...,n there is a unique object A3 in Catmod(SP3), called multiple amalgamation of A1 and $A2^*$ via $A0^*$, written $A3 = A1 +_{A0^*} A2^*$ such that $V_g(A3) = A1$ and $V_{gi}(A3) = A2_i$ for i = 1,...,n.

2. A similar property for morphisms h1, $h2^*$, $h0^*$, defining $h3 = h1 +_{h0^*} h2^*$.

Multiple amalgamation is equivalent to the fact that Catmod transforms multiple pushouts in **SPEC** into multiple pullbacks in **CATCAT** (where multiple pullbacks are the dual construction to multiple pushouts). On the other hand, as a consequence of the fact that if (**SPEC**, Catmod) is a specification frame then Catmod transforms pushouts into pullbacks and initial into terminal objects, we have that:

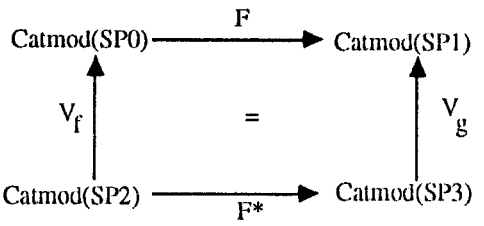
2.5 Theorem

Specification frames have multiple amalgamation.

Multiple extensions can be constructed using multiple amalgamation, in a similar way as extensions can be constructed using amalgamation [EBCO 91]

2.6 Definition

An indexed category SF has multiple extension if for every multiple pushout in **SPEC** and every strongly multiple persistent mapping F: Catmod(SP0) → Catmod(SP1) w.r.t. (f1,...,fn), i.e. for every A1 in Catmod(SP1) and every i $V_{fi}(F(A1)) = A1$, there is a strongly multiple persistent mapping F^*:Catmod(SP2) → Catmod(SP3) w.r.t. (g1,...,gn) s.t. the following diagram commutes



Moreover, if F is a functor then also F^* is a functor.

2.7 Theorem

Specification frames have multiple extension.

3 Basic Specifications and Specification Building Operations

In this section we present the kind of specifications and specification-building operations with which we deal in the rest of the paper. With respect to the specifications, we assume that the logic used for specification is a specification frame, SF = (**SPEC**,Catmod). However, we will not consider that a specification is just an object in **SPEC**. Instead, we will consider that a specification SPC is a pair (SP, C) formed by an object SP in **SPEC** and a set of *constraints* C. Constraints are semantic constructions restricting the possible interpretations of a specification, i.e. a constraint restricts the class of models of a specification to those satisfying the constraint. Different kinds of constraints have been defined and used in the literature [Rei 80, BG 80, SW 82, SW 83, EWT 82, Ehr 81, ON 90, BG 92]. The importance of constraints is a consequence, on the one hand, of the additional expressive power that they provide to specifications and, on the other, because they are the basis of a number of structuring concepts and constructions found in specification languages [BG 80, BG 92, OSC 89].

It can be proved (for details see e.g. [EBCO 91]) that, given a specification frame SF = (**SPEC**,Catmod), the pair SFC = (**SPECC**, CatmodC) is also a specification frame, called induced specification frame with constraints, where **SPECC** is the category of specifications with constraints and CatmodC is the functor that maps every pair (SP, C) to the full subcategory of Catmod(SP) of all models of SP satisfying the constraints in C.

There are mainly two reasons for working with an induced specification frame with constraints instead of with an arbitrary specification frame. The first one is of a methodological nature: we wanted to handle constraints explicitly by one specification-building operation (i.e. **impose**, see below), because some kinds of constraints are a major structuring semantic tool (see [BG80, Rei 80, OSC 89, Bau 91, GB 92]) which, in our opinion, have to be dealt with at the specification language level, rather than considering them embedded in the underlying specification frame. The second reason is of a technical nature: at some points of

the paper we need to consider specifications over *unitary* constraints which can be handled more uniformly if we consider that they already belong to the associated logic of constraints. In particular, an unitary constraint is a constraint that restricts a specification to a single model. Being more precise, given a specification SP and a model A in Catmod(SP), the unitary constraint C_A is defined: $\forall B \in$ Catmod(SP) B satisfies C_A iff $B \equiv A$

The syntax for writing specifications is given by a set of specification expressions over a set of specification-building operations, i.e. a specification expression is a term built by applying a number of specification-building operations to "constant" specification expressions. As a consequence, every specification expression E denotes a basic specification $[E]_{Spec}$. In particular, expressions are defined as follows:

3.1 Definition

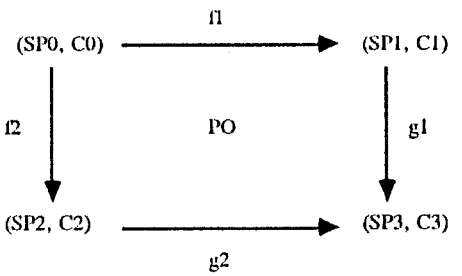
The set of specification expressions over SPC is the least set satisfying:

1. If SP0 $\in$ **SPEC** then SP0 is a specification expression. The specification denoted by SP0 is SP0 together with an empty set of constraints, i.e. $[SP0]_{Spec} = (SP0,\emptyset)$.

2. **impose** C **on** E, where E is a specification expression and C is set of constraints over $[E]_{Spec}$, is a specification expression. In this case $[$**impose** C' **on** E$]_{Spec} = (SP, C \cup C')$, where $[E]_{Spec} = (SP,C)$.

3. **enrich** E **by** f, where f is a morphism from $[E]_{Spec}$ to a given SPC' in **SPECC**, is a specification expression. In this case, $[$**enrich** E **by** f$]_{Spec}$ = SPC'.

4. E1 $+_{(E0, f1, f2)}$ E2, fi : $[E0]_{Spec} \rightarrow [Ei]_{Spec}$ is a morphism in Mor(**SPECC**) ($\forall$ i = 0,..,2) is a specification expression. In this case, $[E1 +_{(E0, f1, f2)} E2]_{Spec} = (SP3,C3)$ defined as the result of the pushout:



where $[Ei]_{Spec} = (SPi,Ci)$ ($\forall$ i = 0,..,2).

Obviously, at the model level, the meaning of these expressions may be defined by considering the model classes of the specifications denoted by the expressions, i.e. $CatmodC([E]_{Spec})$, but we can also define a model level semantics by providing meaning to all the specification-building operations in terms of operations on the model classes:

3.2 Definition

The semantics at the model level for specification expressions is defined as follows:

1. $[SP0]_{Mod}$ = Catmod(SP0)

2. $[$**impose** C' **on** E$]_{Mod} = \{A \in [E]_{Mod} \mid A \models C'\}$.

3. $[$**enrich** E **by** f$]_{Mod}$ = CatmodC(SPC').

4. $[E1 +_{(E0, f1, f2)} E2]_{Mod} = [E1]_{Mod} +_{[E0]_{Mod}} [E2]_{Mod}$ = CatmodC(SP3,C3)

Remarks

1. The two definitions are compatible in the sense that $Catmod_C([E]_{Spec}) = [E]_{Mod}$, for every expression E. This is a consequence of the properties of specification frames.

2. Following the Module Algebra approach [BHK 90] we can consider the specification-building operations as operators of an abstract signature ΣSPEC over two sorts (specifications and morphisms). In this sense, specification expressions are just terms in $T_{\Sigma SPEC}$ and the two semantic definitions $[\_]_{Spec}$ and $[\_]_{Mod}$ can be seen just as the definition of two ΣSPEC-algebras in terms of **SPECC** and $Catmod_C$(**SPECC**), respectively. This fact is used below to define specification expressions over variables, i.e. terms in $T_{\Sigma SPEC}(X)$, and variable substitution.

3. Comparing our specification-building operations, and the ones considered in [SST 90], essentially two differences may be found. The first one is that they consider an operation to close (up to isomorphism) the class of models of a given specification. We do not consider such an operation since we assume (although it is not explicitly stated) that in our framework model classes are already closed up to isomorphism. Actually we think that having the possibility of defining a specification that excludes some isomorphic models is a quite awkward choice, either for the specification frame or for the specification language. The second difference is that they consider a *derive* operation that allows the hiding or forgetting of parts of a specification. Our opinion is that this kind of operation should only be considered in a second layer of a specification language. Being specific, hiding parts of a specification should only be possible through the use of a notion of module with well-defined import and export interfaces describing the "visible" parts of a specification. Actually this is the main feature of modules systems such as the ones found in Extended ML and ACT TWO [ST 89, EM 90].

4 Parameterizations

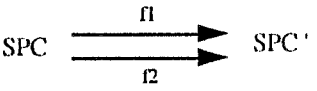
As said in the introduction, following [SST 90] we consider two kinds of parameterizations: parameterized specifications and specifications of parameterized data types. The main difference is methodological: when designing a software system, specifications of parameterized data types are intended to be descriptions of software units that will eventually "exist" in the system; conversely parameterized specifications are user-defined specification-building operations that are just utilized in the specification design phase. A typical example of a specification of parameterized data types may be the ubiquitous STACK[X] specification, describing generic stacks. Obviously, this specification can be used for building arbitrary stack specifications and, in this sense, can be considered as a parameterized specification. But we can better see STACK[X] as the specification of a generic module for building program units implementing arbitrary stacks. A typical example of a parameterized specification can be the specification COMMUT[X] that, given a specification of a data type including a binary operation, yields as result the parameter specification enriched with the commutativity axiom for that operation. For instance, passing as parameter a specification of Groups would produce as result the specification of Abelian Groups. COMMUT[X] can hardly be seen as the description of a software unit.

As in [SST 90], we consider that parameterizations are defined as arbitrary specification expressions over a specification variable. This generalizes a number of previous approaches (e.g. [TWW 78, BG 80, Ehc 82, EM 85]) where only an enrichment of the parameter specification was allowed. In particular, also as in [SST 90], the notation used is the following one: λX:SPC.E[X] denotes a parameterized specification mapping classes of SPC-models

into classes of E[X/SPC]-models, where SPC is a specification with constraints, E[X/SPC] is the expression obtained by substituting in E the variable X by SPC, with $E[X] \in T_{\Sigma SPEC}(\{X\})$ (see remark 3.2.2); similarly, ΠX:SPC.E is the specification of a parameterized data type denoting mappings that associate E[X/SPC]-models to SPC-models.

In the rest of the section we study the semantics of both kinds of parameterizations. Moreover we provide two levels of semantics. At the model level, the definitions follow the intuitions discussed above. At the specification level, the definitions can be seen as a form of abstract syntax for the two parameterization mechanisms. This specification level semantics is needed in section 5 for defining the semantics of parameter passing.

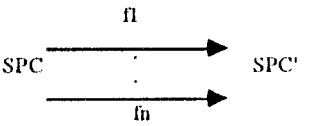
In the most simple approaches to parameterization (e.g.[TWW 82, BG 80, Ehc 82, EKTWW 84]) a parameterized specification is seen as an inclusion of specifications, $SPC \subseteq SPC'$. This is sometimes slightly generalized to arbitrary morphisms f: SPC → SPC'. However, in our context, the semantics at the specification level of a parameterized specification is going to be defined as a set of morphisms from the formal parameter into the result specification. The intuition for this is that, if parameterized specifications are defined by arbitrary specification-building operations, then the formal parameter specification "may occur" several times inside the body specification. A simple example of this is the λ-expression λX:SPC.(enrich X by f)+(enrich X by f'). More realistic examples can be found easily, especially when dealing with higher-order parameterizations. In this example, the specification level semantics is represented by the following diagram:



where SPC' contains two copies of SPC and f1 and f2 are, respectively, two morphisms mapping SPC into each of the copies.

4.1 Definition

Given a parameterized specification λX:SPC.E[X] its semantics at the specification level, $[\lambda X:SPC.E[X]]_{Spec}$, is defined as a triple (SPC, SPC', F), where SPC', usually called the *body specification*, is the specification denoted by E[X/SPC], i.e. $[E[X/SPC]]_{Spec}$ and F is a set of n morphisms, with n≥0, from SPC to SPC', i.e.:



Depending on the form of E (see def. 3.1), $[\lambda X:SPC.E[X]]_{Spec}$ is defined as follows:

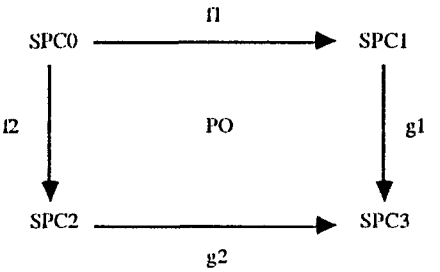
1. If E[X]=SP0 then $[\lambda X:SPC.E[X]]_{Spec}$ = (SPC,SP0,Ø)

2. If E[X]=X then $[\lambda X:SPC.E[X]]_{Spec}$ = (SPC,SPC,{id: SPC → SPC})

3. If E[X]=**impose** C **on** E'[X] and $[\lambda X:SPC.E'[X]]_{Spec}$ = (SPC, (SP',C'), F') then $[\lambda X:SPC.E[X]]_{Spec}$ = (SPC,(SP',C'∪C),{i ∘ f: SPC → (SP',C'∪C)/ f∈F'}) where i denotes the inclusion : (SP',C') → (SP',C'∪C).

4. If E[X] = **enrich** E'[X] **by** f, $[\lambda X:SPC.E'[X]]_{Spec}$ = (SPC, SPC', F') and f: SPC'→ SPC" then $[\lambda X:SPC.E[X]]_{Spec}$ = (SPC,SPC",{f ∘ g: SPC → SPC"/ g∈F'}).

5. If $E[X] = E1[X] +_{(E0[X], f1, f2)} E2[X]$, $[\lambda X{:}SPC.Ei[X]]_{Spec} = (SPC, SPCi, Fi)$ $(0 \le i \le 2)$ and f1 and f2 are morphisms from SPC0 to SPC1 and SPC2, resp., then $[\lambda X{:}SPC.E[X]]_{Spec} = (SPC, SPC3, F3)$, where $F3 = \{g1 \circ h1{:}\ SPC \rightarrow SPC3 /\ h1 \in F1\} \cup \{g2 \circ h2{:}\ SPC \rightarrow SPC3 /\ h2 \in F2\}$) where SPC3, g1 and g2 are defined by the following pushout diagram.



The specification level semantics of ΠX:SPC.E[X] is defined as the one for the corresponding parameterized specification, i.e. $[\Pi X{:}SPC.E[X]]_{Spec} = [\lambda X{:}SPC.E[X]]_{Spec}$

According to the intuitions discussed above, the second level semantics of parameterized specifications is a function mapping model classes into model classes. In particular, given a parameterized specification λX:SPC.E[X], this function can be defined, for every class of SPC-models M, as the evaluation at the model level of the expression E over M.

4.2 Definition

Given a parameterized specification λX:SPC.E[X], its semantics at the model level, $[\lambda X{:}SPC.E[X]]_{Mod}$, is a function from $2^{CatmodC(SPC)}$ to $2^{CatmodC(E[X/SPC])}$ defined for every $M \subseteq CatmodC(SPC)$ as the *evaluation* of E over M, denoted E[X/M] (although the abuse of notation), where:

1. If E[X] = SP0 then E[X/M] = CatmodC(SP0)
2. If E[X] = X then E[X/M] = M
3. If E[X] = impose C on E'[X] then $E[X/M] = \{A \in E'[X/M] / A \models C\}$.
4. If E[X] = enrich E'[X] by f and f is a morphism from E'[X/SPC] to SPC', then $E[X/M] = \{A \in CatmodC(SPC') / V_f(A) \in E'[X/M]\}$.
5. If $E[X] = E1[X] +_{(E0[X], f1, f2)} E2[X]$, then $E[X/M] = E1[X/M] +_{E0[X/M]} E2[X/M]$

It may be noticed that, due to technical problems, we cannot directly define E[X/M] as the evaluation at the model level of the "constant" expression E[X/impose $\{C_M\}$ on SPC], i.e. $[E[X/\text{impose } \{C_M\} \text{ on } SPC]]_{Mod}$, where C_M denotes the constraint that is only satisfied by the models in M. The reason is that if E[X] is (for instance) enrich E'[X] by f, then we know that f is a morphism defined on E'[X/SPC], but not on E[X/impose $\{C_M\}$ on SPC].

4.3 Proposition

Given λX:SPC.E[X], with $[\lambda X{:}SPC.E[X]]_{Spec} = (SPC, SPC', F)$, and $M \subseteq CatmodC(SPC)$ we have:

$$[\lambda X{:}SPC.E[X]]_{Mod}(M) = \{A \in CatmodC(SPC') / \forall f \in F\ V_f(A) \in M\}$$

4.4 Properties of $[\_]_{Mod}$

For every parameterized specification λX:SPC.E[X] we have that:

1. $[\lambda X{:}SPC.E[X]]_{Mod}$ is monotonic, in the sense that for every $M1, M2 \subseteq CatmodC(SPC)$, $M1 \subseteq M2$ implies $[\lambda X{:}SPC.E[X]]_{Mod}(M1) \subseteq [\lambda X{:}SPC.E[X]]_{Mod}(M2)$.

2. In general $[\lambda X{:}SPC.E[X]]_{Mod}$ is not additive, in the sense that for every $M \subseteq CatmodC(SPC)$ $[\lambda X{:}SPC.E[X]]_{Mod}(M) \neq \cup_{A \in M} [\lambda X{:}SPC.E[X]]_{Mod}(\{A\})$.

3. If $[\lambda X{:}SPC.E[X]]_{Spec} = (SPC, SPC', F)$ and $Card(F) \leq 1$ then $[\lambda X{:}SPC.E[X]]_{Mod}$ is additive.

Proof Sketch

The proofs of properties 1 and 3 are trivial. For the second property it is enough to consider the following counter-example. Let SPC and SPC' be the following specifications:

SPC = **sorts** s **opns** a,b

SPC' = **sorts** s' **opns** a',b'

let A and B be two SPC-algebras defined: $A_s = \{0,1\}$, $a_A = 0$, $b_A = 1$ and $B_s = \{0\}$, $a_B = 0 = b_B$ and consider the parameterized specification λX:SPC.X + (enrich X by f) where f is the morphism from SPC to SPC' associating s', a' and b' to s, a and b, respectively.

Now, the algebra C defined $C_s = \{0,1\}$, $C_{s'} = \{0\}$, $a_C = 0$, $b_C = 1$, $a'_C = 0 = b'_C$ is in $F(\{A,B\})$ but not in $F(\{A\}) \cup F(\{B\})$, where $F = [\lambda X{:}SPC.X + (\text{enrich } X \text{ by } f)]_{Mod}$. ■

A similar (though slightly more complex) counter-example can be found in [SST 90]. Now the model level semantics of parameterized data types specifications is defined as the class of all mappings F such that, for every model A, F(A) is in the result of evaluating the parameterization over A.

4.5 Definition

Given a specification of parameterized data types ΠX:SPC.E[X], its semantics at the model level, $[\Pi X{:}SPC.E[X]]_{Mod}$, is the class of all mappings F from CatmodC(SPC) to CatmodC(E[X/SPC]) such that for every $A \in CatmodC(SPC)$, $F(A) \in E[X/\{A\}]$

4.6 Remarks

1. Let $[\Pi X{:}SPC.E[X]]_{Spec}$ be (SPC, SPC', F) if $F \in [\Pi X{:}SPC.E[X]]_{Mod}$ then F is strongly multiple persistent with respect to F, i.e. $\forall A \in CatmodC(SPC)$ and $\forall f \in F$ $V_f \circ F(A) = A$.

2. It may seen also sensible to restrict $[\Pi X{:}SPC.E[X]]_{Mod}$ to the class of "structure-preserving" mappings (i.e. functors) from CatmodC(SPC) to CatmodC(E[X/SPC]), but this restriction has some undesirable consequences that are discussed further in 4.9.

An important issue concerning specifications consists in finding an adequate notion of (internal) correctness. For non-parameterized specification a reasonable and simple correctness condition is consistency, i.e. existence of models. In this sense, a similar condition for specifications of parameterized data types would be asking for $[\Pi X{:}SPC.E[X]]_{Mod} \neq \emptyset$. With respect to parameterized specifications, since they are intended for building specifications, we can base their correctness on their "ability" to build correct specifications. Then, we can define two different correctness conditions: a strong one, if every result of applying the parameterization is consistent, and a weak one, if only some results are consistent.

4.7 Definitions

A parameterized specification $\lambda X{:}SPC.E[X]$ is strongly correct (resp. weakly correct) iff $\forall M \subseteq CatmodC(SPC)$, with $M \neq \emptyset$, $[\lambda X{:}SPC.E[X]]_{Mod}(M) \neq \emptyset$ (resp. $\exists M \subseteq CatmodC(SPC)$ $[\lambda X{:}SPC.E[X]]_{Mod}(M) \neq \emptyset$).

A specification of parameterized data types $\Pi X{:}SPC.E[X]$ is correct iff $[\Pi X{:}SPC.E[X]]_{Mod} \neq \emptyset$.

4.8 Facts

1. $\lambda X{:}SPC.E[X]$ is weakly correct iff $CatmodC(E[X/SPC]) \neq \emptyset$.

2. $\Pi X{:}SPC.E[X]$ is correct iff $\lambda X{:}SPC.E[X]$ is strongly correct.

4.9 Remark

It must be noted that if the model level semantics of specifications of parameterized data types would have been restricted to functor classes then some "reasonable" specifications of this kind would be considered incorrect. This can be seen in the following counter-example:

Let SPC and SPC' be the following first order logic specifications:

SPC =	**sorts** s1	SPC' = SPC +	**sorts**	s2
	opns a,b: s1		**opns**	c, d: s2
			axms	$a = b \Rightarrow c \neq d$
				$a \neq b \Rightarrow c = d$

and let us consider the parameterized data type ΠX: SPC.enrich X by i, where i is the inclusion from SPC to SPC'. This specification may be seen as the loose specification of a family of parameterized data types defining a type s2, with two constants c and d, from a type s1, also with two constants a and b, in such a way that c and d must have a different value if a and b have the same one, and vice-versa. However, there are no functors from Catmod(SPC) to Catmod(SPC'): if we consider the two algebras A and B in Catmod(SPC) defined $A_{s1} = \{0,1\}$, $B_{s1} = \{2\}$ with $a_A = 0$, $b_A = 1$ and $a_B = b_B = 2$, then we have an obvious homomorphism from A to B. However for any A', B' in Catmod(SPC'), such that $V_i(A') = A$ and $V_i(B') = B$, there cannot exist any homomorphism from A' to B'.

4.10 Remark (The Initial Case)

All the results presented in this and the following section can be directly applied to the initial approach to algebraic specification [GTW 78, EM 85]. It is enough to consider that specifications carry an initial or free constraint. In this sense, the results in this paper can be applied to generalize all known results for parameterizations in the initial framework.

5 Parameter Passing

In the previous section we have studied the syntax and semantics for the two kinds of parameterizations, in this section we will study the mechanisms for parameter passing. In particular, this consists in describing the result of applying a parameterization to an actual parameter specification that "matches" the formal parameter, this matching being done (as usual) through a fitting morphism. Note that this is not just evaluating the expression associated to the parameterization over the actual parameter, since this may be meaningless: a similar problem was already discussed for a simpler case after definition 4.2, which was

then solved in a quite ad-hoc manner. In this section we find a more systematic solution by using the "multiple constructions" introduced in this paper, that can be found in section 2.

The result of parameter passing is described at the specification and at the model levels showing compositionality results, namely that the class of models of the resulting specification of a parameter passing operation coincides with the model-level semantics of that operation. It has sometimes been argued that describing parameter passing (and, in general, any other specification building operation) at the specification level is methodologically inadequate, since this means "flattening" (losing the structure of) the given specification. We think that this may be true in some cases but not in this one. The reason is that, in our framework, the "structure" of a specification is defined at two levels: at the higher one, the structure is defined by means of modules, which are not treated in this paper, and at the lower level the structure is defined in terms of the constraints associated to the specification (e.g. see [OSC 89]). In this sense, providing a semantic definition at the specification level is methodologically fully adequate.

5.1 Definition

The syntax of parameter passing for a parameterized specification is given by

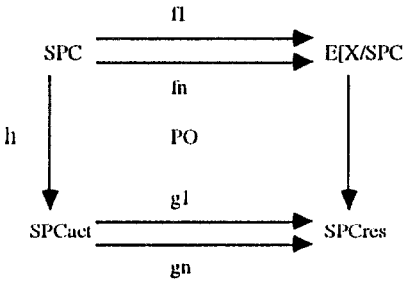
$(\lambda X\text{: SPC.E[X]})\ (SPC_{act})_h$

where λX: SPC.E[X] is the given parameterized specification, SPC_{act} is the actual parameter and h is the parameter passing morphism h: SPC $\rightarrow SPC_{act}$.

For defining the result of parameter passing (at the specification level) of both kinds of parameterizations we use a multiple pushout construction. This construction works as a pushout but taking care that the "multiple occurrences" of the formal parameter in the body of the parameterization are substituted by multiple occurrences of the actual parameter.

5.2 Definition

The result at the specification level of the parameter passing expression $(\lambda X\text{: SPC.E[X]})\ (SPC_{act})_h$ is the specification SPC_{res}, denoted $[(\lambda X\text{: SPC.E[X]})\ (SPC_{act})_h]_{Spec}$, obtained as the result of the following *multiple pushout diagram*:



The model level semantics must be defined in terms of the meaning of the parameterized specification and the actual parameter given. This is done by using multiple amalgamation .

5.3 Definition

The result at the model level of the parameter passing expression $(\lambda X\text{: SPC.E[X]})\ (SPC_{act})_h$, denoted $[(\lambda X\text{: SPC.E[X]})(SPC_{act})_h]_{Mod}$, is defined as the *multiple amalgamation* with respect to the above multiple pushout diagram of the following model classes:

$[(\lambda X: SPC.E[X])(SPC_{act})_h]_{Mod} = [(\lambda X: SPC.E[X])]_{Mod}(M)+_M * CatmodC(SPC_{act})^*$
where $M = V_h(CatmodC(SPC_{act}))$

A compositionality theorem can be proved by showing (by induction on the structure of specification expressions) that $[E[X/SPC]]_{Mod}+_{CatmodC(SPC)}^*$ $CatmodC(SPC_{act})^*$ and $[(\lambda X:SPC.E[X])]_{Mod}(M)+_M * CatmodC(SPC_{act})^*$ coincide.

5.4 Theorem

$CatmodC([(\lambda X: SPC.E[X])\ (SPC_{act})_h]_{Spec}) = [(\lambda X: SPC.E[X])\ (SPC_{act})_h]_{Mod}$

Given the parameter passing expression $(\lambda X: SPC.E[X])\ (SPC_{act})_h$, the following facts hold concerning the correctness and consistency of the specifications involved:

5.5 Facts

1. If $\lambda X: SPC.E[X]$ is strongly correct and SPC_{act} is consistent then SPC_{res} is consistent.

2. If $[(\lambda X: SPC.E[X])]_{Spec} = (SPC, SPC', F)$, $card(F) > 0$ and SPC_{res} is consistent then $\lambda X:SPC.E[X]$ is weakly correct and SPC_{act} is consistent.

3. If $[(\lambda X: SPC.E[X])]_{Spec} = (SPC, SPC', F)$ and $card(F) = 0$ then SPC_{res} is consistent iff SPC' is consistent.

The syntax of parameter passing for specifications of parameterized data types shows the fact that only models are admissible actual parameters. However, not only formal parameter models are acceptable, but also models of any specification matching the formal parameter are considered. Then, to define the result of parameter passing at the specification level, we identify the given model with its specification, including a corresponding unitary constraint.

5.6 Definition

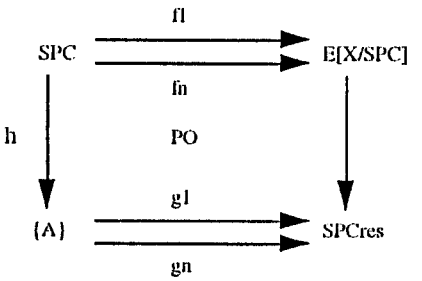
The syntax of parameter passing for a specification of parameterized data types is given by:

$(\Pi X: SPC.E[X])\ (A)_h$

where $A \in CatmodC(SPC_{act})$ and h is teh parameter passing morphism $h : SPC \rightarrow SPC_{act}$.

5.7 Definition

The result at the specification level of the parameter passing expression $(\Pi X: SPC.E[X])\ (A)_h$ is the specification SPC_{res} obtained as the result $[(\Pi X: SPC.E[X])(A)_h]_{Spec}$ of the following multiple pushout diagram:



where {A} denotes the specification **impose** C_A **on** SPC_{act}.

For defining the model level semantics we use the multiple extension construction (see 2.6), allowing to define, for every function mapping formal parameter models into body models, a corresponding extension mapping actual parameter models into result models, in such a way that the multiple occurrences problem is handled adequately.

5.8 Definition

The result at the model level of the parameter passing expression $(\Pi X: SPC.E[X])\ (A)_h$, denoted $[(\Pi X: SPC.E[X])\ (A)_h]_{Mod}$, is the following class of models:

$[(\Pi X:SPC.E[X])(A)_h]_{Mod} = \{F^*_.(A)/\ F^*$ is the multiple extension of an F in $[(\Pi X:SPC.E[X])]_{Mod}\}$

In this context, compositionality can only be ensured if we work in a monomorphic framework. This is reasonable because, as discussed above, specifications of parameterized data types are specifications of true software units. As a consequence, parameter passing would only make sense at the programming language level which is always monomorphic.

5.9 Theorem

1. If $(\Pi X: SPC.E[X])$ is correct and $[(\Pi X: SPC.E[X])\ (A)_h]_{Spec}$ is monomorphic then

   $CatmodC([(\Pi X: SPC.E[X])\ (A)_h]_{Spec}) = [(\Pi X: SPC.E[X])\ (A)_h]_{Mod}$

2. If $\Pi X: SPC.E[X]$ is correct then $(\Pi X: SPC.E[X])\ (A)_h$ is consistent.

6 Conclusions and Further Work

In this paper we have studied in detail the two kinds of parameterization proposed in [SST 90], especially providing mechanisms of parameter passing defined both at the specification and at the model level in a compositional manner. The results obtained not only go beyond the ones obtained in [SST 90] but, at the same time, generalize all previous results for parameterized specifications in the initial case with standard parameter passing.

However an important problem, partly studied in [SST 90], has been left out, namely the definition of higher-order parameterization and parameter passing mechanisms. Recent work in this direction show that it is possible to extend the constructions in this paper to form cocartesian closed categories of parameterized specifications, together with associated cartesian closed categories of models, where all compositionality results would still hold.

7 References

[Bau 91] Baumeister, H.: Unifying Initial and Loose Semantics of Parameterized Specifications in an Arbitrary Institution. Proc. TAPSOFT 91, Brighton. Springer LNCS 493, pp. 103-120, 1991.

[BG 80] Burstall, R.M.; Goguen, J.A.: The semantics of Clear, a specification language, Proc. Copenhagen Winter School on Abstract Software Specification, Springer LNCS 86, pp. 292-332, 1980.

[BGT 91] Burstall, R.M.; Goguen, J.A.; Tarlecki, A.: Some Fundamental Algebraic Tools for the Semantics of Computation, Part 3: Indexed Categories. Theor. Comp. Sc 91 (1991) 239-264.

[BHK 90] Bergstra, J.A.; Heering, J.; Klint, R.: Module Algebra, JACM 37, 2 (1990) 335-372.

[EBCO 91] Ehrig, H.; Baldamus, M.; Cornelius, F.; Orejas, F.: Theory of Algebraic Module Specifications including Behavioural Semantics and Constraints, Proc. AMAST'91, to appear in Springer LNCS 1991

[Ehc 82] Ehrich, H.-D.: On the theory of specification, implementation and parameterization of abstract data types. JACM 29, pp. 209-277, (1982)

[Ehr 81] Ehrig, H.: Algebraic theory of parameterized specifications with requirements, Proc. 6th. CAAP, Springer LNCS 112 (1981) 1-24.

[EKTWW 84] Ehrig; H.; Kreowski, H.-J.; Thatcher, J.; Wagner, E. Wright, J.: Parameter passing in algebraic specification languages. Theor. Comp. Science 28, 45-81, (1984)

[EM 85] Ehrig, H.; Mahr, B.: Fundamentals of Algebraic Specification 1. EATCS Monographs on Theoretical Computer Science, Springer 1985

[EM 90] Ehrig, H.; Mahr, B.: Fundamentals of Algebraic Specification 2. EATCS Monographs on Theoretical Computer Science, Springer (1990)

[EPO 89] Ehrig, H.; Pepper, P.; Orejas, F.: On Recent Trends in Algebraic Specification, Proc. ICALP'89, Springer LNCS 372 (1989), pp. 263-288

[EWT 82] Ehrig, H., Wagner, E.G. Thatcher: J.W. Algebraic constraints for specifications and canonical form results, Institut für Software und Theoretische Informatik, T.U. Berlin Bericht Nr. 82-09, 1982.

[GB 84] Goguen, J.A.; Burstall, R.M.: Introducing institutions. Proc. Logics of Programming Workshop, Carnegie-Mellon. LNCS 164, Springer (1984), 221-256

[GB 92] Goguen, J.A.; Burstall, R.M.: Institutions: abstraact model theory for specification and programming. JACM 39, 1 (1992) 95-146.

[OSC 89] Orejas, F.; Sacristán, V.; Clerici, S.: Development of algebraic specifications with constraints, in 'Categorical Methods in Computer Science - with Aspects from Topology', (H. Ehrig, H. Herrlich, H.-J. Kreowski, G. Preuß, eds.), LNCS 393 (1989)

[Rei 80] Reichel, H.: Initially restricting algebraic theories, Proc. MFCS 80, Springer LNCS 88 (1980), pp. 504-514.

[SST 90] Sannella, D.; Sokolowski, S.; Tarlecki, A.: Toward formal development of programs from algebraic specifications: parameterisation revisited. To appear in Acta Informatica.

[ST 89] Sannella, D.; Tarlecki, A.: Toward formal development of ML programs: foundations and methodology. Proc. TAPSOFT 89. Springer LNCS 352, 375-389 (1989)

[SW 82] Sannella, D; Wirsing, M.: Implementation of parameterised specifications, Proc. 9th ICALP, Springer LNCS 140 (1982) 473-488.

[SW 83] Sannella, D; Wirsing, M. A kernel language for algebraic specification and implementation, Proc. FCT-83, Springer LNCS 158, pp. 413-427, (1983)

[TWW 82] Thatcher, J.W.; Wagner, E.G.; Wright, J.B.: Data type specification: parameterization and the power of specification techniques. Trans. Prog. Lang. and Systems 4 (1982), 711-732

[Wir 86] Wirsing, M.; Structured algebraic specifications: a kernel language. Theor. Comp. Sc. 42, 123-249 (1986)