Toward formal development of programs from algebraic specifications: implementations revisited (Extended abstract)

Donald Sannella[1] and Andrzej Tarlecki[2]

Abstract

The program development process is viewed as a sequence of implementation steps leading from a specification to a program. Based on an elementary notion of refinement, two notions of implementation are studied: constructor implementations which involve a construction "on top of" the implementing specification, and abstractor implementations which additionally provide for abstraction from some details of the implemented specification. These subsume most formal notions of implementation in the literature. Both kinds of implementations satisfy a vertical composition and a (modified) horizontal composition property. All the definitions and results generalise to the framework of an arbitrary institution.

1 Introduction

There has been a lot of interesting work done on notions of refinement (see e.g. [GTW 78], [GB 80], [Ehr 81,82], [EKMP 82], [SW 82], [GM 82], [Gan 83], [Lip 83]). In [SW 83] and then in [ST 85b,86b] we used a very simple notion of specification refinement which seems appropriate for loose specifications: a specification *SP refines to* a specification *SP'*, if every model of *SP'* is a model of *SP*; this extends to a notion of refinement of parameterised specifications. This looks suspiciously oversimplified, especially in comparison with most previous work in this area. In this paper we elaborate on how this simple notion can provide a basis for realistic and non-trivial program development.

Roughly speaking, first we allow an implementation of a specification *SP* by another specification *SP'* to consist of a "program" or construction written in terms of *SP'* to compute the functions specified in *SP*. This subsumes most previous notions of implementation in the literature, e.g. [GTW 78], [Ehr 82], [EKMP 82] and [SW 82]. Then we incorporate ideas concerning *behavioural equivalence* of algebras as discussed in [GGM 76], [Rei 81], [GM 82], [ST 86b] (and elsewhere), by allowing the construction to deliver a result which realises *SP* not "exactly" but only up to an equivalence on algebras. This subsumes the notions of implementation in [Ehr 81], [GM 82], [Sch 82] and [BMPW 86]. These notions extend to parameterised specifications as before.

In order to be useful for stepwise and modular program development, implementations should compose *vertically* and *horizontally* [GB 80]. The simple notion of refinement enjoys both of these properties. The first extended notion composes vertically and satisfies a (modified) horizontal composition property; similar results for the second notion hold only under certain additional conditions.

We present these ideas in the framework of partial algebras [BrW 82]. This is mainly to take advantage of the reader's intuition, since all of the main definitions and results as well as methodological remarks may be directly restated in the framework of an arbitrary institution [GB 84]. This means that they can be used to develop programs from specifications in a wide variety of logical systems. Thus, a user of the presented program development methodology may choose the logical system which is most suited to his particular task. Moreover, different logical systems may be most suitable at different stages of the development of even a single program, for example when developing an efficient imperative program from a high-level algebraic specification. We enable this by allowing specifications to be implemented by specifications in a different institution using what we call a *semi-institution morphism* [Tar 86].

[1]Department of Artificial Intelligence, University of Edinburgh and Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh

[2]Institute of Computer Science, Polish Academy of Sciences, Warsaw

Unfortunately, for lack of space we are not able to cover this topic here; the interested reader should consult [ST 87] for a detailed treatment of this subject as well as for all the proofs, examples and full discussion which we are forced to omit here.

2 Algebraic preliminaries

Due to lack of space we omit the definitions of the following standard algebraic notions: signature (Σ), signature morphism (σ), the category **Sign** of signatures with initial (empty) signature $\Sigma_\emptyset$; partial Σ-algebra A, (closed) Σ-subalgebra, (weak) Σ-homomorphism, the category $\mathbf{PAlg}(\Sigma)$ of partial Σ-algebras; the σ-reduct functor $\_|_\sigma\colon \mathbf{PAlg}(\Sigma') \to \mathbf{PAlg}(\Sigma)$ for any signature morphism $\sigma\colon \Sigma \to \Sigma$; terms t, equations $\forall X.t = t'$, definedness formulae $D(t)$, partial (first-order) sentences φ, and their translations ($\sigma(t)$, etc.) under signature morphisms. All these definitions may be found in [ST 87] and elsewhere. We write $A \models \varphi$ to denote that the algebra A satisfies φ, defined in the usual way (generalised to classes of algebras and sets of sentences as usual).

For any signature Σ and $S \subseteq sorts(\Sigma)$, we say that a Σ-algebra A is *reachable on S* if it contains no proper Σ-subalgebra with carriers of sorts not in S the same as in A. In other words, every element of A is the value of a Σ-term with variables of sorts not in S (for some valuation). Notice that any Σ-algebra A contains exactly one Σ-subalgebra which is reachable on S and has carriers of sorts not in S the same as in A, denoted $\mathcal{R}_S(A)$. We omit qualification by S in these definitions if $S = sorts(\Sigma)$.

Let $A \in \mathbf{PAlg}(\Sigma)$. A *congruence on A* is an equivalence relation $\equiv\ \subseteq |A| \times |A|$ such that for any $f\colon s_1, \ldots, s_n \to s$ in Σ and $a_1, b_1 \in |A|_{s_1}, \ldots, a_n, b_n \in |A|_{s_n}$, if $a_1 \equiv_{s_1} b_1, \ldots, a_n \equiv_{s_n} b_n$ and $f_A(a_1, \ldots, a_n)$ and $f_A(b_1, \ldots, b_n)$ are defined, then $f_A(a_1, \ldots, a_n) \equiv_s f_A(b_1, \ldots, b_n)$. The quotient of an algebra by a congruence is defined as usual.

3 Specifications and refinement

We are not going to formally define precisely what specifications are; they are just finite syntactic objects of some kind. Every specification describes a certain signature and a class of algebras over this signature. This semantics is made explicit using two mappings which assign to each specification SP a signature $Sig[SP] \in |\mathbf{Sign}|$ and a class $Mod[SP] \subseteq |\mathbf{PAlg}(Sig[SP])|$ of $Sig[SP]$-algebras. Algebras in $Mod[SP]$ are called *models of SP*. We call a specification *consistent* if it has at least one model.

This rather general description covers high-level user-oriented loose specifications admitting non-isomorphic models as well as low-level detailed specifications or even programs which for us are just very tight specifications. We adopt a purely model-theoretic view here and stop the analysis of the notion of a program at this level. Any application of the methodology we outline would require some further syntactic constraints on the notion of a program.

Definition 1 *For any signature Σ, $\mathbf{Spec}(\Sigma)$ denotes the collection of all Σ*-specifications, *i.e. specifications SP such that $Sig[SP] = \Sigma$, preordered by the inclusion of model classes. For any two specifications SP1 and SP2, a* specification morphism *$\sigma\colon SP1 \to SP2$ is a signature morphism $\sigma\colon Sig[SP1] \to Sig[SP2]$ such that for any model $A2 \in Mod[SP2]$, $A2|_\sigma \in Mod[SP1]$.*

We assume that $\mathbf{Spec}(\Sigma)$ contains at least *basic specifications*. That is, given a signature Σ and a (finite, recursive, r.e.) set Φ of Σ-sentences, $\langle \Sigma, \Phi \rangle$ is a specification with:

$$\begin{aligned} &Sig[\langle \Sigma, \Phi \rangle] = \Sigma \\ &Mod[\langle \Sigma, \Phi \rangle] = \{A \in \mathbf{PAlg}(\Sigma) \mid A \models \Phi\} \end{aligned}$$

If the sentences are all (universally quantified) equations or definedness formulae we call $\langle \Sigma, \Phi \rangle$ an *equational specification.*

Specification-building operations are used to put together little specifications in nice ways to make progressively bigger ones [BG 77]. Any specification-building operation, given a list of argument specifications, yields a result specification; semantically, a specification-building operation is a function on classes of algebras. The only assumption we make about these functions is that they are monotonic; intuitively, less restrictive argument specifications yield a less restrictive result. Specification languages like CLEAR [BG 77,80] may be viewed just as sets of such operations plus some syntactic sugar.

Example 1 (translate) [ST 86a] Given a specification SP and signature morphism σ: $Sig[SP] \rightarrow \Sigma'$, **translate** SP **by** σ is a specification with semantics defined as follows:

$$Sig[\textbf{translate } SP \textbf{ by } \sigma] = \Sigma'$$
$$Mod[\textbf{translate } SP \textbf{ by } \sigma] = \{A' \in \mathbf{PAlg}(\Sigma') \mid A'|_\sigma \in Mod[SP]\} \qquad \Box$$

Translate is actually a *family* of specification-building operations,

$$\textbf{translate} = \{\textbf{translate}_{\sigma:\Sigma\rightarrow\Sigma'}\colon \mathbf{Spec}(\Sigma) \rightarrow \mathbf{Spec}(\Sigma')\}_{\sigma\in\mathbf{Sign}}$$

For any specification-building operation ω we will write ω: $\mathbf{Spec}(\Sigma) \rightarrow \mathbf{Spec}(\Sigma')$, meaning that ω takes Σ-specifications to Σ'-specifications. Note that we have tacitly assumed that ω is a unary operation; to simplify the presentation we make the same assumption throughout when convenient.

A specification language usually provides a way for the user to define his own specification-building operations, i.e. a mechanism for constructing *parameterised specifications*. There are different approaches to parameterised specifications; in this paper we use the approach of [ST 86a].

Semantically, any parameterised specification can be viewed as a function taking any specification over a given parameter signature Σ_{par} to a specification over a result signature Σ_{res}. Syntactically, we write a parameterised specification as a λ-expression, $\lambda X\colon \Sigma_{par}.SP_{res}[X]$, where X is an identifier and $SP_{res}[X]$ is a Σ_{res}-specification built using specification-building operations which may involve X as a variable denoting a Σ_{par}-specification. For any Σ_{par}-specification SP, $(\lambda X\colon \Sigma_{par}.SP_{res}[X])(SP)$ is a specification with semantics defined (essentially as β-conversion) as follows:

$$Sig[(\lambda X\colon \Sigma_{par}.SP_{res}[X])(SP)] = \Sigma_{res}$$
$$Mod[(\lambda X\colon \Sigma_{par}.SP_{res}[X])(SP)] = Mod[SP_{res}[SP/X]]$$

We sometimes write $(\lambda X\colon \Sigma_{par}.SP_{res}[X])\colon \mathbf{Spec}(\Sigma_{par}) \rightarrow \mathbf{Spec}(\Sigma_{res})$ to indicate the parameter and result signatures explicitly.

The programming discipline of stepwise refinement suggests that a program (which is a specification) be evolved from a high-level specification by working gradually via a series of successively more detailed lower-level intermediate specifications. A formalisation of this approach requires a precise definition of the concept of refinement.

Definition 2 *Given two specifications SP and SP' such that $Sig[SP] = Sig[SP']$, we say that SP* refines to *SP', written $SP \rightsquigarrow SP'$, if $Mod[SP'] \subseteq Mod[SP]$.*
Given two parameterised specifications P and P' with the same parameter signature Σ_{par}, we say that P refines to *P', written $P \rightsquigarrow P'$, if for any Σ_{par}-specification SP, $P(SP) \rightsquigarrow P'(SP)$.*

Intuitively, $SP \rightsquigarrow SP'$ if SP' incorporates more design decisions than SP.

An important issue for any notion of refinement is whether refinements can be composed *vertically* ($SP \rightsquigarrow SP'$ and $SP' \rightsquigarrow SP''$ implies $SP \rightsquigarrow SP''$) and *horizontally* ($P \rightsquigarrow P'$ and $SP \rightsquigarrow SP'$ implies $P(SP) \rightsquigarrow P'(SP')$) [GB 80]. The above notion of refinement has both these properties since specification-building operations are monotonic. These properties allow large structured specifications to be refined in a gradual and modular fashion.

The development of a program from a specification consists of a series of refinement steps $SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n$, where SP_0 is the original high-level specification and SP_n is a program. Vertical composability guarantees the correctness of SP_n with respect to its specification SP_0. This views each of the specifications $SP_0, \ldots, SP_n$ as a single indivisible entity. If, however, we decompose any of them using a parameterised specification, say $SP_k = P(SP)$, then the further developments of P and of SP may proceed separately. Horizontal composability guarantees that the results of these developments may always be combined to give a refinement of SP_k and so of SP_0 as well. Of course, these (sub)developments may themselves involve further decomposition.

4 Constructors and implementations

The simple notion of refinement is mathematically elegant but perhaps a bit oversimplified from a practical point of view. In the sequel, we will develop notions of implementation built on top of this simple notion of refinement which are more suited to practical use. We start with a notion of implementation which involves a construction from the implementing specification to the implemented specification.

What is a construction? Model-theoretically, the characteristic feature of a construction is that it transforms an algebra over one signature to yield another algebra over a (possibly different) signature. Thus, we can identify a construction κ with a function[3] $\kappa \colon \mathbf{PAlg}(\Sigma) \to \mathbf{PAlg}(\Sigma')$. This determines a specification-building operation denoted (ambiguously) by the same symbol. We call specification-building operations of this kind *constructors*.

Definition 3 *A* constructor *determined by a function* $\kappa \colon \mathbf{PAlg}(\Sigma) \to \mathbf{PAlg}(\Sigma')$ *is a specification-building operation* $\kappa \colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$, *where for any Σ-specification SP,* $Sig[\kappa(SP)] = \Sigma'$ *and* $Mod[\kappa(SP)] = \{\kappa(A) \mid A \in Mod[SP]\}$.

Fact 1 *Constructors are monotonic, preserve consistency of specifications, and are closed under composition.* □

Example 2 (derive) For any Σ'-specification SP' and signature morphism $\sigma \colon \Sigma \to \Sigma'$, the semantics of the specification **derive from** SP' **by** σ is as follows:

$$Sig[\textbf{derive from } SP' \textbf{ by } \sigma] = \Sigma$$
$$Mod[\textbf{derive from } SP' \textbf{ by } \sigma] = \{A|_\sigma \mid A \in Mod[SP']\}$$

The **derive** specification-building operations (one for each $\sigma \colon \Sigma \to \Sigma'$) are constructors determined by the corresponding reduct functors $\_|_\sigma$. Intuitively, **derive** can be used to hide and/or rename some of the sorts and operations of a specification. □

Example 3 (restrict) For any Σ-specification SP and set $S \subseteq sorts[\Sigma]$ of sorts, the semantics of the specification **restrict** SP **on** S is as follows:

$$Sig[\textbf{restrict } SP \textbf{ on } S] = \Sigma$$
$$Mod[\textbf{restrict } SP \textbf{ on } S] = \{\mathcal{R}_S(A) \mid A \in Mod[SP]\}$$

The **restrict** specification-building operations (one for each Σ and $S \subseteq sorts[\Sigma]$) are constructors determined by the corresponding restrict functors $\mathcal{R}_S$. **Restrict** is used to remove "junk", i.e. to restrict to the reachable part of algebras. □

[3]From the category-theoretic point of view, it is natural to assume that this is a functor (all our examples are) but since we do not use the morphism part in this paper we take this simplified view here.

Example 4 (quotient) For any Σ-specification SP and congruence $\sim$ on ground Σ-terms, the semantics of the specification **quotient** SP **wrt** $\sim$ is as follows:

$$\begin{aligned}&\mathit{Sig}[\textbf{quotient } SP \textbf{ wrt } \sim] = \Sigma\\ &\mathit{Mod}[\textbf{quotient } SP \textbf{ wrt } \sim] = \{A/{\sim} \mid A \in \mathit{Mod}[SP]\}\end{aligned}$$

The **quotient** specification-building operations (one for each Σ and $\sim$ on Σ-terms) are constructors determined by the corresponding quotient functors $\_/{\sim}$. Intuitively, **quotient** is used to identify the values of certain terms; usually the congruence $\sim$ is presented via a set of equations. □

Example 5 (extend) If we have a signature morphism $\sigma\colon \Sigma \to \Sigma'$ then constructors from $\mathbf{Spec}(\Sigma)$ to $\mathbf{Spec}(\Sigma')$ will be called *synthesizing constructors along σ*. The intuition is that they just build new stuff on top of the existing algebras without forgetting anything. One standard way to define such a synthesizing constructor is using the free extension.

Namely, for any signature morphism $\sigma\colon \Sigma \to \Sigma'$ and equational Σ'-specification SP', there is a free functor $\mathbf{F}_\sigma\colon \mathbf{PAlg}(\Sigma) \to \mathit{Mod}[SP']$ (the left adjoint to the reduct functor $\_|_\sigma\colon \mathit{Mod}[SP'] \to \mathbf{PAlg}(\Sigma)$). That this functor always exists is a well-known fact. For any Σ-specification SP, **extend** SP **to** SP' **via** σ is a specification defined as follows:

$$\begin{aligned}&\mathit{Sig}[\textbf{extend } SP \textbf{ to } SP' \textbf{ via } \sigma] = \Sigma'\\ &\mathit{Mod}[\textbf{extend } SP \textbf{ to } SP' \textbf{ via } \sigma] = \{\mathbf{F}_\sigma(A) \mid A \in \mathit{Mod}[SP]\}\end{aligned}$$

Note that SP may be an arbitrary specification here, not necessarily equational. In general $\mathbf{F}_\sigma$ does not have to preserve all the properties required by SP (so σ was not required to be a specification morphism $\sigma\colon SP \to SP'$) although it does preserve ground equations deducible from SP. □

Non-example (translate) The **translate** specification-building operation defined in the last section is *not* a constructor. Consider for example any $\sigma\colon \Sigma_\emptyset \to \Sigma$ where Σ is non-empty or any $\sigma'\colon \Sigma \to \Sigma'$ which is non-injective on sorts. □

Definition 4 *A synthesizing constructor* κ: Spec$(\Sigma) \to$ Spec(Σ') *is* persistent along *a signature morphism* $\sigma\colon \Sigma \to \Sigma'$*, written* κ: Spec$(\Sigma) \xrightarrow{\sigma}$ Spec(Σ')*, if* $\kappa\colon \mathbf{PAlg}(\Sigma) \to \mathbf{PAlg}(\Sigma')$ *is (strongly) persistent with respect to σ, i.e. for any Σ-algebra A, $\kappa(A)|_\sigma = A$.*

Example 6 (amalgamated union) Given two persistent constructors $\kappa1$: $\mathbf{Spec}(\Sigma) \xrightarrow{\sigma1} \mathbf{Spec}(\Sigma1)$ and $\kappa2$: $\mathbf{Spec}(\Sigma) \xrightarrow{\sigma2} \mathbf{Spec}(\Sigma2)$, let

$$\begin{tikzcd}\Sigma \arrow[r, "\sigma1"] \arrow[d, "\sigma2"'] & \Sigma1 \arrow[d, "\sigma1'"] \\ \Sigma2 \arrow[r, "\sigma2'"'] & \Sigma'\end{tikzcd}$$

be a pushout in **Sign**. For any Σ-algebra A, define $\kappa(A)$ to be the unique Σ'-algebra such that $\kappa(A)|_{\sigma1'} = \kappa1(A)$ and $\kappa(A)|_{\sigma2'} = \kappa2(A)$. $\kappa(A)$ is well-defined since $\kappa1(A)|_{\sigma1} = A = \kappa2(A)|_{\sigma2}$. Thus, we have defined a function $\kappa\colon \mathbf{PAlg}(\Sigma) \to \mathbf{PAlg}(\Sigma')$. We denote this function and the corresponding synthesizing constructor (along $\sigma1;\sigma1' = \sigma2;\sigma2'$) by $\kappa1 + \kappa2$; if any doubts may arise, we add $\sigma1, \sigma2$ as subscripts to $+$. Intuitively, $\kappa1 + \kappa2$ "puts together" the constructions $\kappa1$ and $\kappa2$. The assumption of persistency guarantees that this is possible. (See the notion of amalgamated sum in [PB 85] and [EM 85].) □

Fact 2 *If* $\kappa1$: Spec$(\Sigma) \xrightarrow{\sigma1}$ Spec$(\Sigma1)$ *and* $\kappa2$: Spec$(\Sigma) \xrightarrow{\sigma2}$ Spec$(\Sigma2)$ *are persistent constructors then* $\kappa1 + \kappa2$: Spec$(\Sigma) \xrightarrow{\sigma}$ Spec(Σ') *is a persistent constructor along* $\sigma =_{def} \sigma1;\sigma1' = \sigma2;\sigma2'$. □

Example 7 (translation of a constructor) There is another operator on constructors connected with the pushout in **Sign**. Namely, reconsider the pushout diagram of example 6 and suppose $\kappa 1$: $\mathbf{Spec}(\Sigma) \xrightarrow{\sigma 1} \mathbf{Spec}(\Sigma 1)$ is a persistent constructor. Then for any $A2 \in \mathbf{PAlg}(\Sigma 2)$, define $\sigma 2(\kappa 1)(A2)$ to be the unique Σ'-algebra such that $\sigma 2(\kappa 1)(A2)|_{\sigma 1'} = \kappa 1(A2|_{\sigma 2})$ and $\sigma 2(\kappa 1)(A2)|_{\sigma 2'} = A2$. Thus we have defined a function $\sigma 2(\kappa 1)$: $\mathbf{PAlg}(\Sigma 2) \to \mathbf{PAlg}(\Sigma')$ which we call the *translation of* $\kappa 1$ *along* $\sigma 2$. We use the same notation and terminology to refer to the corresponding synthesizing constructor (along $\sigma 2'$). Notice that $\sigma 2(\kappa 1)$ is persistent. Intuitively, $\sigma 2(\kappa 1)$ performs $\kappa 1$ on the "Σ part" of $\Sigma 2$-algebras and leaves the other components unchanged. Notice that the translation of a constructor is a more elementary operation than the amalgamated union. Namely, using the notation of example 6, $\kappa 1 + \kappa 2 = \kappa 2;\sigma 2(\kappa 1) = \kappa 1;\sigma 1(\kappa 2)$. □

Definition 5 (constructor implementation) *A specification SP is implemented by a specification SP' via a constructor* κ: $\mathbf{Spec}(Sig[SP']) \to \mathbf{Spec}(Sig[SP])$, *written* $SP \underset{\kappa}{\rightsquigarrow\!\!>} SP'$, *if* $SP \rightsquigarrow\!\!> \kappa(SP')$.

Intuitively speaking, if we want to evaluate a function in *SP*, we are able to do this provided we can evaluate any function in *SP'* since the constructor κ puts together functions in *SP'* to obtain all functions in *SP*. In this sense, κ may be viewed as a program parameterised by the (possibly not yet executable) specification *SP'*.

Notice that, using the constructors introduced in examples 2-5 above, we can reduce many of the notions of implementation in the literature (e.g. [GTW 78], [Ehr 82], [EKMP 82], [SW 82]) to the one above. For example, the implementation notion of [EKMP 82] assumes that κ is the composition of **extend**, **derive**, **restrict** and **quotient** constructors (in that order).

Our definition of constructor implementation resembles the notion of implementation given in [Ehr 81] for single algebras. In [Ehr 81], A is implemented by B via a construction F if A is (isomorphic to) a quotient of a subalgebra of $F(B)$. When generalising to loose specifications, the requirement that *some* quotient of *some* subalgebra of $F(B)$ be isomorphic to A may be regarded as a construction only if the subalgebra and quotient are taken uniformly on all models B of the implementing specification. If we do not require uniformity then this amounts to a non-constructive step which will be fully subsumed by the notion of abstractor implementation defined in section 5. There are even closer similarities with the notion of implementation of (parameterised) specifications in [Lip 83]; see section 6.1 for details.

Theorem 1 (vertical composition) *If* $SP \underset{\kappa}{\rightsquigarrow\!\!>} SP'$ *and* $SP' \underset{\kappa'}{\rightsquigarrow\!\!>} SP''$ *then* $SP \underset{\kappa';\kappa}{\rightsquigarrow\!\!>} SP''$. □

Notice that since $\kappa';\kappa$ is an acceptable constructor, there is no reason to require that it has (or may be transformed to) the same form as either κ or κ'. In general this will not be the case. However, in some special cases it turns out that such normal form theorems may be obtained, often under some additional assumptions about the specifications involved (see e.g. [Ehr 81], [EKMP 82], [SW 82], [EWT 83], [Ore 83]). It seems to us that the requirement that the composition of constructors must be forced into some given normal form corresponds to requiring programs to be written in a rather restrictive programming language which does not provide sufficiently powerful modularisation facilities for the job. In some situations, putting a constructor into a normal form can be viewed as an optimization process.

The following simple fact allows us to mechanically strip off outermost constructors if the specification we want to implement happens to be built in this way.

Fact 3 *For any constructor* κ: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ *and* Σ*-specification SP,* $\kappa(SP) \underset{\kappa';\kappa}{\rightsquigarrow\!\!>} SP'$ *provided that* $SP \underset{\kappa'}{\rightsquigarrow\!\!>} SP'$. □

An interesting special case of this is the amalgamated union of specifications.

Definition 6 *For any two specification morphisms $\sigma 1$: $SP \to SP1$ and $\sigma 2$: $SP \to SP2$, the* amalgamated union of $SP1$ and $SP2$, *written $SP1 + SP2$ (decorated with subscripts $SP, \sigma 1, \sigma 2$ on $+$ if necessary), is a specification with semantics defined as follows:*

$$\begin{aligned} &Sig[SP1 + SP2] = \Sigma' \\ &Mod[SP1 + SP2] = Mod[\textbf{translate } SP1 \textbf{ by } \sigma 1'] \cup Mod[\textbf{translate } SP2 \textbf{ by } \sigma 2'] \end{aligned}$$

where the following diagram is a pushout in **Sign**:

$$\begin{tikzcd} Sig[SP] \arrow[r, "\sigma 1"] \arrow[d, "\sigma 2"'] & Sig[SP1] \arrow[d, "\sigma 1'"] \\ Sig[SP2] \arrow[r, "\sigma 2'"'] & \Sigma' \end{tikzcd}$$

Theorem 2 *If $SP1 \underset{\kappa 1}{\rightsquigarrow} SP$ and $SP2 \underset{\kappa 2}{\rightsquigarrow} SP$ where both $\kappa 1$: $\mathbf{Spec}(Sig[SP]) \xrightarrow{\sigma 1} \mathbf{Spec}(Sig[SP1])$ and $\kappa 2$: $\mathbf{Spec}(Sig[SP]) \xrightarrow{\sigma 2} \mathbf{Spec}(Sig[SP2])$ are persistent constructors, then $SP1 + SP2 \underset{\kappa 1 + \kappa 2}{\rightsquigarrow} SP$.* □

This theorem allows us to implement the independent components of a specification separately and then combine their implementations provided that they do not affect the common part.

In the above theorem we required $\kappa 1$ and $\kappa 2$ to be persistent on all $Sig[SP]$-algebras as in the definition of the amalgamated union of constructors. However, in this context (as well as in similar situations in the sequel) it is sufficient to require that $\kappa 1$ and $\kappa 2$ are persistent only on models of SP (which may be easier to achieve in practice). Of course formally, $\kappa 1 + \kappa 2$ is then only a constructor on $Mod[SP]$ rather than on $\mathbf{PAlg}(Sig[SP])$ since it may be undefined on some $Sig[SP]$-algebras.

Theorem 3 *Let*

$$\begin{tikzcd} \Sigma \arrow[r, "\sigma 1"] \arrow[d, "\sigma 2"'] & \Sigma 1 \arrow[d, "\sigma 1'"] \\ \Sigma 2 \arrow[r, "\sigma 2'"'] & \Sigma' \end{tikzcd}$$

be a pushout in **Sign**, *$\kappa 1$: $\mathbf{Spec}(\Sigma) \xrightarrow{\sigma 1} \mathbf{Spec}(\Sigma 1)$ be a persistent constructor, and $SP1$,$SP2$ be $\Sigma 1$- and $\Sigma 2$-specifications respectively. If $SP1 \underset{\kappa 1}{\rightsquigarrow}$* **derive from** *$SP2$* **by** *$\sigma 2$ then $SP1 + SP2 \underset{\sigma 2(\kappa 1)}{\rightsquigarrow} SP2$.*
□

This gives another way of decomposing a specification and implementing the components separately. Namely, we implement one component using (a part of) the other and then we can proceed with the implementation of the other component.

Summing up, the development process using this notion of implementation would consist of a sequence of steps $SP_0 \underset{\kappa_1}{\rightsquigarrow} SP_1 \underset{\kappa_2}{\rightsquigarrow} \cdots \underset{\kappa_n}{\rightsquigarrow} SP_n$. Intuitively, SP_0, SP_1 etc. do not "grow" as happens when we use the simple refinement notion, where this development would look like:

$$SP_0 \rightsquigarrow \kappa_1(SP_1) \rightsquigarrow \cdots \rightsquigarrow \kappa_1(\ldots \kappa_n(SP_n)\ldots)$$

Using constructor implementations, we gradually reduce the specification by implementing its parts. Our goal is to end up with an empty specification over the empty signature, i.e. $SP_n = \langle \Sigma_\emptyset, \emptyset \rangle$. Then, the composition of constructors $\kappa_n; \cdots ; \kappa_1$ forms a program which implements SP_0.

5 Abstractors and implementations

It is often possible to abstract away from some of the details of the user's original specification without violating the real intention behind it. This is the idea behind the specification technique known in software engineering as *abstract model specification* [LB 77], in which the user defines in a more or less concrete fashion a model which gives the desired results with the intention that any program giving the same answers is acceptable. This theme has been discussed in [GGM 76], [Rei 81], [GM 82], [Kam 83], [ST 85a] and elsewhere; the idea goes back (at least) to work on automata theory in the 1950's [Moo 56].

To formalize these ideas we will consider another class of specification-building operations called abstractors. Intuitively, any equivalence relation on Σ-algebras determines a specification-building operation which relaxes interpretation of any Σ-specification SP by admitting as a model any Σ-algebra which is equivalent to a model of SP.

Definition 7 *An* abstractor determined by *an equivalence relation* $\equiv \subseteq \mathbf{PAlg}(\Sigma) \times \mathbf{PAlg}(\Sigma)$ *is a specification-building operation* $\alpha_{\equiv}$: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ *where for any Σ-specification SP,*

$$Sig[\alpha_{\equiv}(SP)] = \Sigma$$
$$Mod[\alpha_{\equiv}(SP)] = \{A \in \mathbf{PAlg}(\Sigma) \mid \exists A' \in Mod[SP].A \equiv A'\}$$

In the sequel we will omit the subscript $\equiv$ when there is no danger of confusion. Also, if α is known we denote the abstraction equivalence *which determines it by $\equiv_{\alpha}$.*

Fact 4 *Abstractors are monotonic, idempotent, and preserve and reflect consistency of specifications.* □

In general, abstractors are not closed under composition. This fact is neither surprising nor disturbing; we will not in fact have occasion to compose abstractors.

Example 8 (observational abstraction) For any Σ-specification SP and set W of ground Σ-terms, the semantics of the specification **abstract** SP **wrt** W is as follows [SW 83]:

$$Sig[\mathbf{abstract}\ SP\ \mathbf{wrt}\ W] = \Sigma$$
$$Mod[\mathbf{abstract}\ SP\ \mathbf{wrt}\ W] = \{A \in \mathbf{PAlg}(\Sigma) \mid \exists A' \in Mod[SP].A \equiv_W A'\}$$

where for any two algebras $A, A' \in \mathbf{PAlg}(\Sigma)$, $A \equiv_W A'$ iff:

- for all $t \in W$, $A \models D(t)$ iff $A' \models D(t)$, and
- for all $s \in sorts(\Sigma)$ and all $t, t' \in W_s$, $A \models t = t'$ iff $A' \models t = t'$.

Intuitively, W is the set of Σ-terms which represent computations the user is allowed to perform. We do not want to distinguish between algebras in which all these computations give the same results. A similar idea in the context of concurrent processes appears in [deNH 84]. □

Example 9 (behavioural abstraction) An important special case of observational abstraction is behavioural abstraction. For any Σ-specification SP and set $OBS \subseteq sorts(\Sigma)$ of sorts, the semantics of the specification **behaviour** SP **wrt** OBS is as follows [SW 83], [ST 86a], [ST 86b]:

$$Sig[\mathbf{behaviour}\ SP\ \mathbf{wrt}\ OBS] = \Sigma$$
$$Mod[\mathbf{behaviour}\ SP\ \mathbf{wrt}\ OBS] = \{A \in \mathbf{PAlg}(\Sigma) \mid \exists A' \in Mod[SP].A \equiv_{OBS} A'\}$$

where the equivalence $\equiv_{OBS}$ is just $\equiv_W$ for W the set of all ground Σ-terms of sorts in OBS. Intuitively, OBS is the set of external sorts, visible to the user. □

Definition 8 (abstractor implementation) *A Σ-specification SP is implemented by a Σ'-specification SP'* wrt *an abstractor α:* $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ *via a constructor κ:* $\mathbf{Spec}(\Sigma') \to \mathbf{Spec}(\Sigma)$, *written $SP \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} SP'$, if $\alpha(SP) \rightsquigarrow \kappa(SP')$.*

If in the above definition, α is behavioural abstraction, then intuitively speaking we are implementing the behaviour of SP rather than SP itself. This subsumes the notions of implementation in [GM 82], [Sch 82] and [BMPW 86]. Notice that the abstractor α cannot be chosen arbitrarily; the choice depends on the specification SP and the context in which it is to be used. If α abstracts too much then the implementation will be useless — for example if $\equiv$ is the total equivalence on $\mathbf{PAlg}(\Sigma)$ then $SP \overset{\alpha\equiv}{\underset{\kappa}{\rightsquigarrow}} SP'$ for *any* SP' and constructor κ: $\mathbf{Spec}(Sig[SP']) \to \mathbf{Spec}(Sig[SP])$.

Suppose $SP \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} SP'$ and $SP' \overset{\alpha'}{\underset{\kappa'}{\rightsquigarrow}} SP''$. We would like to be able to conclude that $SP \overset{\alpha}{\underset{\kappa';\kappa}{\rightsquigarrow}} SP''$. According to the above argument we assume that α was chosen appropriately for the context in which SP is to be used and so we do not want to change it even when composing implementations. In general, there is no hope for such a result. If α' is too "liberal", there is no reason to expect that κ transforms any $\alpha'(SP')$-model to a model of $\alpha(SP)$. However, the following theorem does hold:

Theorem 4 (vertical composition) *If $SP \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} SP'$ and $SP' \overset{\alpha'}{\underset{\kappa'}{\rightsquigarrow}} SP''$ then $SP \overset{\alpha}{\underset{\kappa';\kappa}{\rightsquigarrow}} SP''$ provided* κ preserves the abstraction equivalences, *i.e. for any two algebras $A1, A2 \in$ $\mathbf{PAlg}(Sig[SP'])$ if $A1 \equiv_{\alpha'} A2$ then $\kappa(A1) \equiv_\alpha \kappa(A2)$.* □

A methodological conclusion from this theorem is that the development process should proceed as follows: starting from a specification SP considered in a context for which an abstractor α is appropriate, we (abstractor) implement SP, say $SP \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} SP'$. The next step should be to establish the appropriate abstractor up to which SP' may be considered by "pushing $\equiv_\alpha$ through κ". Namely, this should be the abstractor determined by the equivalence $\kappa^{-1}(\equiv_\alpha)$ where for $A, A' \in \mathbf{PAlg}(Sig[SP'])$, $A\ \kappa^{-1}(\equiv_\alpha)\ A'$ iff $\kappa(A) \equiv_\alpha \kappa(A')$. Then, we can proceed with the development of SP' in the context of the abstractor determined by $\kappa^{-1}(\equiv_\alpha)$. (Actually, any equivalence finer than $\kappa^{-1}(\equiv_\alpha)$ will do.) Similar ideas in the context of concurrent processes appear in [Lar 86].

Corollary 1 *If $SP_0 \overset{\alpha_1}{\underset{\kappa_1}{\rightsquigarrow}} \cdots \overset{\alpha_n}{\underset{\kappa_n}{\rightsquigarrow}} SP_n$ and $\equiv_{\alpha_2} \subseteq \kappa_1^{-1}(\equiv_{\alpha_1})$ and $\cdots$ and $\equiv_{\alpha_n} \subseteq \kappa_{n-1}^{-1}(\equiv_{\alpha_{n-1}})$ then $SP_0 \overset{\alpha_1}{\underset{\kappa_n;\cdots;\kappa_1}{\rightsquigarrow}} SP_n$.* □

Note that in practice, it is often convenient to sharpen the above results. They hold if the constructors preserve the equivalences between models of the appropriate specifications (e.g. in the vertical composition theorem it is sufficient that $\kappa(A1) \equiv_\alpha \kappa(A2)$ for any $A1 \in \mathbf{PAlg}(Sig[SP'])$ and $A2 \in Mod[SP']$ such that $A1 \equiv_{\alpha'} A2$).

In the rest of this section, we show that vertical composition and the above methodological remarks may work in practice. On one hand, the constructors we have introduced do preserve appropriate (observational) equivalences; and on the other hand, we show how to push standard observational equivalences in a satisfactory way through the constructors we have defined.

Lemma 1 (derive) *For any signature morphism σ: $\Sigma1 \to \Sigma2$ and set W of ground $\Sigma2$-terms, $\mathcal{D}_\sigma^{-1}(\equiv_W) = \equiv_{\sigma(W)}$, where $\mathcal{D}_\sigma$:* $\mathbf{Spec}(\Sigma2) \to \mathbf{Spec}(\Sigma1) =_{def} \lambda X$: $\Sigma2$. **derive from** X **by** σ. □

Lemma 2 (restrict) *For any signature Σ, $S \subseteq sorts(\Sigma)$ and set W of ground Σ-terms, $A \equiv_W \mathcal{R}_S(A)$ for all Σ-algebras A, where $\mathcal{R}_S$:* $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma) =_{def} \lambda X$: Σ. **restrict** X **on** S. □

The above lemma gives directly a characterisation of the result of pushing observational equivalence through **restrict** constructors. Moreover, it directly implies that **restrict** steps may be skipped if we use abstractor implementations.

Corollary 2 *Under the assumptions of lemma 2, $\mathcal{R}_S^{-1}(\equiv_W) = \equiv_W$.* □

Corollary 3 *Under the assumptions of lemma 2, if α is the abstractor determined by $\equiv_W$, then for any Σ-specifications SP and SP', $SP \underset{R_S}{\overset{\alpha}{\rightsquigarrow}} SP'$ implies $SP \underset{id}{\overset{\alpha}{\rightsquigarrow}} SP'$.* □

It is worth pointing out that the above corollary also allows us to throw out **restrict** steps "in the middle" of the development process (provided that the intermediate equivalence used in this step satisfies the assumptions of lemma 2). This means that corollary 2 becomes superfluous since instead of using it to push equivalences through **restrict** steps we can just skip these steps entirely.

The situation with **quotient** steps is similar although we need slightly more restrictive assumptions (see [ST 87] for details).

Definition 9 *For any signature morphism $\sigma\colon \Sigma \to \Sigma'$, constructor $\kappa\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ and sets W and W' of ground Σ- and ground Σ'-terms respectively, κ is* observably sufficiently complete *(*wrt *W, W') if for any term $t' \in W'$, either for any $A \in \mathbf{PAlg}(\Sigma)$, $\kappa(A) \not\models D(t')$ or there exists a term $t \in W$ such that for any $A \in \mathbf{PAlg}(\Sigma)$, $\kappa(A) \models t' = \sigma(t)$.*

Typically, we will consider sets W and W' such that observable sufficient completeness is a weaker condition than sufficient completeness, which corresponds to the case where W' is the set of all ground Σ'-terms of the sorts $\sigma(S)$ for $S =_{def} sorts(\Sigma)$ and W is the set of all ground Σ-terms.

Definition 10 *For any signature morphism $\sigma\colon \Sigma \to \Sigma'$, constructor $\kappa\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ and set W of ground Σ-terms, κ is* observably persistent *(*wrt *W) if for all terms $t1, t2 \in W$ of the same sort and anv $A \in \mathbf{PAlg}(\Sigma)$, $\kappa(A) \models \sigma(t1) = \sigma(t2)$ iff $A \models t1 = t2$ and $\kappa(A) \models D(\sigma(t1))$ iff $A \models D(t1)$.*

Notice that observable persistency is a weaker condition than the standard persistency.

Lemma 3 (synthesize) *For any signature morphism $\sigma\colon \Sigma \to \Sigma'$ which is injective on sorts, constructor $\kappa\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ and sets W and W' of ground Σ- and Σ'-terms respectively, if κ is observably sufficiently complete wrt W, W' and observably persistent wrt W then $\kappa^{-1}(\equiv_{W'}) \supseteq \equiv_W$. Moreover, if in addition W is a minimal set such that observable sufficient completeness holds then $\kappa^{-1}(\equiv_{W'}) = \equiv_W$.* □

As remarked already, constructor implementation using the **derive**, **restrict**, **quotient** and **extend** constructors subsumes many of the notions of implementation in the literature. The above lemmas imply that the extension of any of these notions to a corresponding notion of abstractor implementation goes through smoothly.

Lemma 4 (amalgamated union) *Let $\kappa1\colon \mathbf{Spec}(\Sigma) \xrightarrow{\sigma1} \mathbf{Spec}(\Sigma1)$ and $\kappa2\colon \mathbf{Spec}(\Sigma) \xrightarrow{\sigma2} \mathbf{Spec}(\Sigma2)$ be persistent constructors, $W, W1, W2$ be sets of ground Σ-, $\Sigma1$- and $\Sigma2$-terms respectively such that $\kappa1$ is observably sufficiently complete wrt $W, W1$ and $\kappa2$ is observably sufficiently complete wrt $W, W2$. Recall that $\kappa =_{def} \kappa1 + \kappa2\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$, where*

$$\begin{tikzcd}
\Sigma \arrow[r, "\sigma1"] \arrow[d, "\sigma2"'] & \Sigma1 \arrow[d, "\sigma1'"] \\
\Sigma2 \arrow[r, "\sigma2'"'] & \Sigma'
\end{tikzcd}$$

is a pushout in **Sign**, *is a persistent synthesizing constructor (along $\sigma1;\sigma1' = \sigma2;\sigma2'$) such that for $A \in \mathbf{PAlg}(\Sigma)$, $\kappa(A)$ is the unique Σ'-algebra such that $\kappa(A)|_{\sigma1'} = \kappa1(A)$ and $\kappa(A)|_{\sigma2'} = \kappa2(A)$. Under these assumptions, κ is observably sufficiently complete wrt W, W' where $W' =_{def} \sigma1'(W1) \cup \sigma2'(W2)$.* □

Corollary 4 *Under the assumptions of lemma 4, $\kappa^{-1}(\equiv_{W'}) \supseteq \equiv_W$.* □

Lemma 5 (translation of a constructor) *Consider again the pushout diagram from lemma 4. Let $W, W1, W2$ be sets of ground Σ-, $\Sigma1$- and $\Sigma2$-terms respectively, and let* $\kappa1$: $\mathbf{Spec}(\Sigma) \xrightarrow{\sigma1} \mathbf{Spec}(\Sigma1)$ *be a persistent constructor. If $\kappa1$ is observably sufficiently complete wrt $W, W1$ and $\sigma2(W) \subseteq W2$ then* $\sigma2(\kappa1)$: $\mathbf{Spec}(\Sigma2) \rightarrow \mathbf{Spec}(\Sigma')$ *is observably sufficiently complete wrt $W2, W'$ where $W' = \sigma1'(W1) \cup \sigma2'(W2)$.* □

Corollary 5 *Under the assumptions of lemma 5, $\sigma2(\kappa1)^{-1}(\equiv_{W'}) \supseteq \equiv_{W2}$.* □

6 Parameterisation and implementations

In the same way as the simple notion of refinement on specifications gave rise to a notion of refinement for parameterised specifications, the definitions of constructor and abstractor implementation extend to notions of constructor and abstractor implementation for parameterised specifications.

6.1 Parameterisation and constructor implementations

Definition 11 *For any parameterised specification P:* $\mathbf{Spec}(\Sigma_{par}) \rightarrow \mathbf{Spec}(\Sigma_{res})$ *and specification-building operation ω:* $\mathbf{Spec}(\Sigma_{res}) \rightarrow \mathbf{Spec}(\Sigma)$, *$\omega(P)$ is a parameterised specification defined by $\omega(P) =_{def} \lambda X\colon \Sigma_{par}.\omega(P(X))$:* $\mathbf{Spec}(\Sigma_{par}) \rightarrow \mathbf{Spec}(\Sigma)$.

Definition 12 (constructor implementation) *For any parameterised specifications with a common parameter signature P:* $\mathbf{Spec}(\Sigma_{par}) \rightarrow \mathbf{Spec}(\Sigma)$ *and P':* $\mathbf{Spec}(\Sigma_{par}) \rightarrow \mathbf{Spec}(\Sigma')$ *and constructor κ:* $\mathbf{Spec}(\Sigma') \rightarrow \mathbf{Spec}(\Sigma)$, *$P$* is implemented by *$P'$* via *$\kappa$, written $P \underset{\kappa}{\leadsto} P'$, if $P \leadsto \kappa(P')$.*

This subsumes the notion of implementation of parameterised specifications in [SW 82]. It resembles the one in [Lip 83], where a parameterised specification is a (strongly) persistent functor. According to [Lip 83], P is implemented by P' via a construction F (another persistent functor, obtained by composing certain specification-building operations) if there is some P'' and (persistent) natural transformations i: $P'' \rightarrow P';F$ and s: $P'' \rightarrow P$ such that i and s are componentwise injective and surjective respectively. In our framework, this corresponds roughly to an implementation via the composition of a persistent constructor, a **restrict** step and a **quotient** step (in that order). Although there are several other definitions of implementation of parameterised specifications in the literature (see e.g. [EK 82], [GM 82] and [Gan 83]) it is difficult to compare them with ours because our definition extends the definition for the non-parameterised case in the usual way that a relation is extended from elements to functions (that is, pointwise). In contrast, [EK 82] defines implementation of parameterised specifications by comparing their bodies and then proves that this implies our notion of implementation. This is arguably preferable from the point of view of proving correctness of implementations but we prefer to adopt the natural definition and treat the problem of proving correctness separately.

Theorem 5 (vertical composition) *For any parameterised specifications P, P', P'' with common parameter signature Σ_{par}, if $P \underset{\kappa}{\leadsto} P'$ and $P' \underset{\kappa'}{\leadsto} P''$ then $P \underset{\kappa';\kappa}{\leadsto} P''$.* □

As in fact 3, we can strip off outermost constructors from parameterised specifications:

Fact 5 *For any parameterised specifications P and P' and constructor κ on the result signature of P, $\kappa(P) \underset{\kappa';\kappa}{\leadsto} P'$ provided that $P \underset{\kappa'}{\leadsto} P'$.* □

Constructor implementations do not compose horizontally. In fact, the standard formulation of the horizontal composition property is not even well-formed in this case. Namely, if P: $\mathbf{Spec}(\Sigma_{par}) \rightarrow \mathbf{Spec}(\Sigma_{res})$ is a parameterised specification, SP is a Σ_{par} specification and $SP \underset{\kappa}{\leadsto} SP'$, then in general $Sig[SP'] \neq \Sigma_{par}$ and so $P(SP')$ is not even well-defined. However:

Theorem 6 **(horizontal composition)** *Given a parameterised specification P with parameter signature Σ_{par} and a Σ_{par}-specification SP, if $P \underset{\kappa}{\rightsquigarrow} P'$ and $SP \underset{\mu}{\rightsquigarrow} SP'$ then $P(SP) \underset{\kappa}{\rightsquigarrow} P'(\mu(SP))$.*
□

Although this is not horizontal composition as formulated in [GB 80], it is perfectly adequate for our purposes. It guarantees that in the case of a specification formed by applying a parameterised specification P to a Σ-specification SP, the developments of P and SP may proceed independently and the results be successfully combined. If $P \underset{\kappa_1}{\rightsquigarrow} P_1 \underset{\kappa_2}{\rightsquigarrow} \cdots \underset{\kappa_n}{\rightsquigarrow} P_n$ and $SP \underset{\mu_1}{\rightsquigarrow} SP_1 \underset{\mu_2}{\rightsquigarrow} \cdots \underset{\mu_m}{\rightsquigarrow} SP_m$ then $P(SP) \underset{\kappa_n;\cdots;\kappa_1}{\rightsquigarrow} P_n((\mu_m;\cdots;\mu_1)(SP_m))$. We aim at reducing the parameter specification to the empty specification and the parameterised specification to the identity. If $SP_m = \langle\Sigma_\emptyset, \emptyset\rangle$ and $P_n = \lambda X\colon \Sigma.X$ then the composition of constructors $\mu_m;\cdots;\mu_1;\kappa_n;\cdots;\kappa_1$ implements $P(SP)$.

6.2 Parameterisation and abstractor implementations

Definition 13 **(abstractor implementation)** *For any parameterised specifications with a common parameter signature P: $\mathbf{Spec}(\Sigma_{par}) \to \mathbf{Spec}(\Sigma)$ and P': $\mathbf{Spec}(\Sigma_{par}) \to \mathbf{Spec}(\Sigma')$, abstractor α: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ and constructor κ: $\mathbf{Spec}(\Sigma') \to \mathbf{Spec}(\Sigma)$, P is implemented by P'* wrt *α via κ, written $P \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'$, if $\alpha(P) \rightsquigarrow \kappa(P')$.*

Theorem 7 **(vertical composition)** *For any parameterised specifications P, P', P'' with common parameter signature Σ_{par}, if $P \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'$ and $P' \overset{\alpha'}{\underset{\kappa'}{\rightsquigarrow}} P''$ then $P \overset{\alpha}{\underset{\kappa';\kappa}{\rightsquigarrow}} P''$ provided that κ preserves the abstraction equivalences.* □

Applicability of this result in program development requires proving that the constructors we use preserve the appropriate abstraction equivalences. For this, lemmas 1-5 of section 5 are applicable just as in the non-parameterised case.

Unfortunately, the horizontal composition theorem for abstractor implementations does not hold, even in the form suggested by the horizontal composition theorem for constructor implementations; parameter specifications cannot in general be abstracted from since parameterised specifications can make essential use of non-observable parts of the parameter. One way to circumvent this is to restrict attention to parameterised specifications which use their arguments in an abstract way, so that if we change the argument to an equivalent one we get a result which is equivalent.

Definition 14 *Let α: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ be an abstractor. We say that two Σ-specifications $SP1$ and $SP2$ are* α-equivalent *if $Mod[\alpha(SP1)] = Mod[\alpha(SP2)]$.*

Theorem 8 **(horizontal composition)** *If $P \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'$ and $SP \overset{\alpha'}{\underset{\kappa'}{\rightsquigarrow}} SP'$ then $P(SP) \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'(\kappa'(SP'))$ provided that P* preserves α'-equivalence, *i.e. for any specifications $SP1, SP2$ over the (common) parameter signature of P and P', $P(SP1)$ and $P(SP2)$ are α-equivalent whenever $SP1$ and $SP2$ are α'-equivalent.* □

The requirement that P preserves α'-equivalence in the above theorem is guaranteed in either of the following three cases:

1. P has the form $\lambda X\colon \Sigma.SP1[\alpha'(X)]$, i.e. P abstracts from its argument before using it.

2. P is built entirely from constructors which preserve the relevant abstraction equivalences.

3. The abstractor α' is trivial, i.e. for any specification SP, $Mod[\alpha'(SP)] = Mod[SP]$.

The last case amounts to the following:

Corollary 6 *If $P \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'$ and $SP \underset{\kappa'}{\rightsquigarrow} SP'$ then $P(SP) \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'(\kappa'(SP'))$.* □

A constructor implementation $SP \underset{\kappa'}{\rightsquigarrow} SP'$ is an abstractor implementation $SP \overset{\alpha'}{\underset{\kappa'}{\rightsquigarrow}} SP'$ where the abstractor α' is trivial. Notice however that when we push the corresponding equivalence through κ' and the constructors used in the further implementation of SP', the resulting abstraction equivalences may determine non-trivial abstractors again and so the use of techniques of abstractor implementations may be essential further on.

7 Concluding remarks

A number of important problems connected with the ideas presented here remain to be considered. First, we do not discuss here any methods for proving correctness of refinements; methods for proving theorems in specifications, especially in the context of observational abstraction [ST 86a,86b], are relevant to this problem. This would be especially important in the case of parameterised specifications.

There is a large body of technical work in the literature on different specific notions of implementation. Viewed in our approach, each of these notions corresponds to a restriction on the choice of constructors and abstractors which may be used. We have tried to unify and generalise the many different notions of implementation in the literature. This quest for generality yields a uniform framework in which we can compare different approaches. We can investigate which of the problems encountered under different notions of implementation are inherent to the very concept of what an implementation should be and which are just technicalities caused by the imposed restrictions, and conversely, which results and properties are consequences of such restrictions and which are inherent to the nature of implementations. We have not yet tried to pursue this line of investigation in a systematic manner.

According to our definition, any inconsistent specification refines any specification over the same signature. But if we succeed in refining a specification to a program then the original specification must have been consistent. This means that checking consistency is not necessary to ensure correctness of the development process. However, an inconsistent specification is a blind alley. On the other hand, even a consistent specification may have no computable model and so we cannot in general avoid blind alleys in program development anyway.

In what we have presented here, constructors are just functions rather than actual pieces of programs in the usual sense. We did not give any particular syntax for defining constructors. It would be interesting to develop a programming language which would provide facilities for defining and composing constructors (this would probably require restricting the notion of constructor we use, as implied in section 3). A good starting point seems to be Standard ML [Mil 85] with modules [MacQ 85], where constructors could be defined as Standard ML functors (i.e. parameterised modules).

Acknowledgements
Many of the ideas in this paper evolved in close collaboration with Martin Wirsing. Thanks to Oliver Schoett for many relevant discussions, to Hartmut Ehrig for his criticism which stimulated us to write these ideas down, and to an anonymous referee who directed our attention to [Lip 83]. Thanks to Teresa for (gastronomic) care. This work was supported by grants from the Alvey Directorate and the Polish Academy of Sciences.

8 References

[**BMPW 86**] Broy, M., Möller, B., Pepper, P. and Wirsing, M. Algebraic implementations preserve program correctness. *Science of Computer Programming 7*, pp. 35-53.

[**BrW 82**] Broy, M. and Wirsing, M. Partial abstract types. *Acta Informatica 18* pp. 47-64.

[BG 77] Burstall, R.M. and Goguen, J.A. Putting together theories to make specifications. *Proc. 5th Intl. Joint Conf. on Artificial Intelligence*, Cambridge.

[BG 80] Burstall, R.M. and Goguen, J.A. The semantics of Clear, a specification language. *Proc. of Advanced Course on Abstract Software Specifications*, Copenhagen. Springer LNCS 86, pp. 292-332.

[deNH 84] de Nicola, R. and Hennessy, M.C.B. Testing equivalences for processes. *Theoretical Computer Science 34*, pp. 83-133.

[Ehr 81] Ehrich, H.-D. On realization and implementation. *Proc. 10th Intl. Symp. on Mathematical Foundations of Computer Science*, Strbske Pleso, Czechoslovakia. Springer LNCS 118.

[Ehr 82] Ehrich, H.-D. On the theory of specification, implementation, and parametrization of abstract data types. *Journal of the Assoc. for Computing Machinery 29* pp. 206-227.

[EKMP 82] Ehrig, H., Kreowski, H.-J., Mahr, B. and Padawitz, P. Algebraic implementation of abstract data types. *Theoretical Computer Science 20* pp. 209-263.

[EM 85] Ehrig, H. and Mahr, B. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science, Springer.

[EWT 83] Ehrig, H., Wagner, E.G. and Thatcher, J.W. Algebraic specifications with generating constraints. *Proc. 10th Intl. Colloq. on Automata, Languages and Programming*, Barcelona. Springer LNCS 154, pp. 188-202.

[Gan 83] Ganzinger, H. Parameterized specifications: parameter passing and implementation with respect to observability. *TOPLAS 5*, 3 pp. 318-354.

[GGM 76] Giarratana, V., Gimona, F. and Montanari, U. Observability concepts in abstract data type specification. *Proc. 5th Intl. Symp. on Mathematical Foundations of Computer Science*, Gdansk. Springer LNCS 45.

[GB 80] Goguen, J.A. and Burstall, R.M. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical report CSL-118, SRI International.

[GB 84] Goguen, J.A. and Burstall, R.M. Introducing institutions. *Proc. Logics of Programming Workshop* (E. Clarke and D. Kozen, eds.), Carnegie-Mellon University. Springer LNCS 164, pp. 221-256.

[GM 82] Goguen, J.A. and Meseguer, J. Universal realization, persistent interconnection and implementation of abstract modules. *Proc. 9th Intl. Colloq. on Automata, Languages and Programming*, Aarhus. Springer LNCS 140, pp. 265-281.

[GTW 78] Goguen, J.A., Thatcher, J.W. and Wagner, E.G. An initial algebra approach to the specification, correctness, and implementation of abstract data types. *Current Trends in Programming Methodology, Vol. 4: Data Structuring* (R.T. Yeh, ed.), Prentice-Hall, pp. 80-149.

[Kam 83] Kamin, S. Final data types and their specification. *TOPLAS 5*, 1 pp. 97-121.

[Lar 86] Larsen, K. Context-dependent bisimulation between processes. Ph.D. thesis, Dept. of Computer Science, Univ. of Edinburgh.

[Lip 83] Lipeck, U. Ein algebraischer Kalkül für einer strukturierten Entwurf von Datenabstraktionen. Ph.D. thesis, Abteilung Informatik, Universität Dortmund.

[LB 77] Liskov, B.H. and Berzins, V. An appraisal of program specifications. Computation Structures Group memo 141-1, Laboratory for Computer Science, MIT.

[MacQ 85] MacQueen, D.B. Modules for Standard ML. *Polymorphism 2*, 2.

[Mil 85] Milner, R.G. The Standard ML core language. *Polymorphism 2*, 2.

[Moo 56] Moore, E.F. Gedanken-experiments on sequential machines. In: *Automata Studies* (C.E. Shannon and J. McCarthy, eds.), Princeton Univ. Press, pp. 129-153.

[**Ore 83**] Orejas, F. Characterizing composability of abstract implementations. *Proc. Intl. Conf. on Foundations of Computation Theory*, Borgholm, Sweden. Springer LNCS 158, pp. 335-346.

[**PB 85**] Parisi-Presicce, F. and Blum, E.K. The semantics of shared submodules specifications. *Proc. 10th Colloq. on Trees in Algebra and Programming*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Berlin. Springer LNCS 185, pp. 359-373.

[**Rei 81**] Reichel, H. Behavioural equivalence - a unifying concept for initial and final specification methods. *Proc. 3rd Hungarian Computer Science Conference*, Budapest, pp. 27-39.

[**ST 85a**] Sannella, D.T. and Tarlecki, A. Some thoughts on algebraic specification. *Proc. 3rd Workshop on Theory and Applications of Abstract Data Types*, Bremen. Springer Informatik-Fachberichte Vol. 116, pp. 31-38.

[**ST 85b**] Sannella, D.T. and Tarlecki, A. Program specification and development in Standard ML. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 67-77.

[**ST 86a**] Sannella, D.T. and Tarlecki, A. Specifications in an arbitrary institution. Report CSR-184-85, Dept. of Computer Science, Univ. of Edinburgh; to appear in *Information and Control.*

[**ST 86b**] Sannella, D.T. and Tarlecki, A. On observational equivalence and algebraic specification. Report CSR-172-84, Dept. of Computer Science, Univ. of Edinburgh; to appear in *Journal of Computer and Systems Sciences.*

[**ST 87**] Sannella, D.T. and Tarlecki, A. Toward formal development of programs from algebraic specifications: implementations revisited (full version). Research report, Dept. of Computer Science, Univ. of Edinburgh (to appear).

[**SW 82**] Sannella, D.T. and Wirsing, M. Implementation of parameterised specifications (extended abstract). *Proc. 9th Intl. Colloq. on Automata, Languages and Programming*, Aarhus. Springer LNCS 140, pp. 473-488.

[**SW 83**] Sannella, D.T. and Wirsing, M. A kernel language for algebraic specification and implementation (extended abstract). *Proc. Intl. Conf. on Foundations of Computation Theory*, Borgholm, Sweden. Springer LNCS 158, pp. 413-427.

[**Sch 82**] Schoett, O. A theory of program modules, their specification and implementation (extended abstract). Report CSR-155-83, Dept. of Computer Science, Univ. of Edinburgh.

[**Tar 86**] Tarlecki, A. Software-system development — an abstract view. *Information Processing '86.* North-Holland, pp. 685-688.